

国际信息工程先进技术译丛

WILEY

基于FSM和Verilog HDL的数字电路设计

FSM based digital design using Verilog HDL

[英] 彼德·明斯（Peter Minns） 伊恩·艾利奥特（Ian Elliott） 著

姚世扬 译



机械工业出版社
CHINA MACHINE PRESS

国际信息工程先进技术译丛

基于 FSM 和 Verilog HDL 的数字电路设计

[英] 彼德·明斯（Peter Minns）
伊恩·艾利奥特（Ian Elliott） 著
姚世扬 译



机 械 工 业 出 版 社

北京市版权局著作权合同登记 图字：01 - 2015 - 1927 号。

图书在版编目（CIP）数据

基于 FSM 和 Verilog HDL 的数字电路设计/（英）明斯（Minns，P.），（英）艾利奥特（Elliott，I.）著；姚世扬译. —北京：机械工业出版社，2016. 5

（国际信息工程先进技术译丛）

书名原文：FSM based digital design using Verilog HDL

ISBN 978-7-111-53292-7

Ⅰ. ①基…　Ⅱ. ①明…②艾…③姚…　Ⅲ. ①数字电路 - 电路设计　Ⅳ. ①TN79

中国版本图书馆 CIP 数据核字（2016）第 058305 号

机械工业出版社（北京市百万庄大街 22 号　邮政编码 100037）

策划编辑：徐明煜　责任编辑：徐明煜

责任校对：陈　越　封面设计：马精明

责任印制：乔　宇

北京铭成印刷有限公司印刷

2016 年 5 月第 1 版第 1 次印刷

169mm × 239mm · 23. 25 印张 · 420 千字

0001—2500 册

标准书号：ISBN 978-7-111-53292-7

定价：120. 00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务	网络服务
服务咨询热线：010-88361066	机 工 官 网：www. cmpbook. com
读者购书热线：010-68326294	机 工 官 博：weibo. com/cmp1952
010-88379203	金 书 网：www. golden-book. com
封面无防伪标均为盗版	教育服务网：www. cmpedu. com

本书介绍了基于有限状态机（FSM）的数字电路硬件设计，通过结合工程案例来展示 FSM 是如何融入其中的。同时，本书还运用硬件描述语言 Verilog HDL，通过编写可执行和仿真的代码，让读者从实际应用的角度获得一个完整的数字电路的设计思路。

本书从设计方法，到编程语言，比较系统地介绍了数字电路的硬件设计，并结合实际案例进行详细的剖析。读者能够从本书中学到完整的设计思路，并可以借鉴或整合到自己的方案中，极大地方便了相关高校学生与专业人士的学习和运用。

译　者　序

本书由英国纽卡斯尔诺森比亚大学微电子与通信工程专业的 Peter Minns 先生和 Ian Elliott 先生合著。本人阅读后，认为书中大部分内容能为数字电路系统设计提供一些启发，甚至解决的方案。因此，产生了把这本书翻译成中文，在国内出版的念头。

数字电路的硬件设计，特别是涉及 FPGA 的片内设计已经发展了几十年。市面上有不少这方面的书籍，其中大部分都是根据一些开发板来介绍某类芯片的应用，并附带说明工具软件的使用方法等，虽然对 FPGA 硬件设计有一定帮助，但是大多缺乏一定的深度。本书从设计方法，到编程语言，比较系统地介绍了数字电路的硬件设计，并结合实际案例进行详细的剖析。读者能够从本书中学到完整的设计思路，并可以借鉴或整合到自己的方案中，极大地方便了相关高校学生与专业人士的学习和运用。

如今正是国家大力发展物联网、智慧城市，鼓励科学创新、技术突破的时代。这些都离不开硬件平台的设计和搭建。希望本书的推出，能够为致力于从事硬件开发的朋友们提供一个良好的参考来源和学习途径。

由于本书的翻译全部由本人独立完成，其中会有不少错误、疏漏或者不足之处，在此恳请广大读者及时提出并指正。同时希望借此机会，能够和大家沟通交流，共同学习和进步。

姚世扬

原书前言

本书主要介绍的是在数字系统中如何设计并运用有限状态机（Finite State Macine，FSM），其中包括利用微处理器、微控制器，以及FSM直接控制的存储单元等不同方法进行设计的案例和系统，同时也包含了一些在数字系统设计中经常遇到的情况。因此这里的重点是让读者对有限状态机有一个全面的认识，并掌握在什么情况下使用它以及如何使用它。

Verilog HDL近年来得到了广泛的运用，本书也对其进行了详细的介绍，许多设计案例都是运用它来描述和验证的。除了用Verilog描述逻辑门和布尔代数方程以外，本书专门用一章的篇幅介绍了硬件描述语言在所谓行为级的应用，它指的是通过使用Verilog语言的一些高级功能进行系统设计。

此外，本书中有一个章节介绍了独热编码技术，运用这种技术设计的FSM，更多地出现在现场可编程门阵列（Fidd Programmable Gate Array，FPGA）芯片中，例如动态存储访问（Dynamic Memory Access，DMA）控制器和数据检测系统等。本书还用一章介绍了异步（事件触发）FSM，它不需要时钟驱动，对可靠性要求较高的系统比较适用。关于佩特里（Petri）网络，即并行数字FSM技术，也专门用一章来进行讲述。

在数字系统发展的过程中，微控制器一直被用来控制系统的输入和输出，同时还被用来处理模拟信号。现在，使用本书介绍的技巧和方法作为一种设计辅助，基于状态机的方案可以通过比较固定的模式来实现，即状态图。一旦设计出状态图，工程师便可以直接使用它推导系统的布尔代数方程，也可以根据其流程直接编写Verilog硬件描述语言代码。一些外围设备，例如存储单元、地址计数器或者比较器等，也可以通过布尔代数方程来定义它们的操作，或者使用Verilog语言去描述它们的行为。

本书主要适用于电子和通信工程专业大学本科最后一年的学生，也可用于那些想快速掌握如何使用状态机来设计系统的研究生和工程师们。本书的读者应掌握数字电路基础知识，例如逻辑门电路、布尔代数等。具体章节规划如下：

前3章是帮助读者学习并掌握同步状态机的一些重要的基本概念。排版方式和课堂笔记比较类似，已经作为诺森比亚大学本科最后一年的课件使用了很多年，并取得了良好的反馈。其内容涵盖了状态机设计和综合的基本要素。从第4章开始，书面排版将和一般书籍一样，不过这并不影响其连贯性，读者仍然可以像阅读普通书籍一样来学习前3章的内容。

下面将详细地阐述各个章节所涵盖的内容。

第1章介绍了状态机的基本概念，其中包含米利（Mealy）状态机和摩尔（Moore）状态机这两个主要形态的区别，同步状态机（时钟驱动）和异步状态机（事件驱动）的概念，状态图以及如何使用状态图来表示系统的时序行为及输入和输出的状态等。随后介绍了几个代表性的例子，来帮助读者更好地理解如何使用状态机以达到某个具体的设计目的。

第2章主要对外接的硬件设备的应用进行了阐述，着重介绍了如何用状态机来控制它们。其中包括如何通过使用外接计时器产生等待状态，如何控制模-数转换器（ADC）、存储器件等。这些基于状态机的系统级设计理念，可能在其他类似的书中是不多见的。

第3章是课件部分的延续，排版和前两章类似。主要介绍了如何使用T触发器和D触发器来进行状态表的综合，以及系统初始化的方法。

第4章介绍同步（时钟驱动）状态机，并带有仿真结果。这一章主要是向读者展现一些常见的实用案例，例如数字波形发生器和串行异步收发模块等。

第5章介绍了基于“独热编码”技术的同步状态机，其中包括动态存储访问（DMA）控制器和串行数据检测系统等。

第6章介绍了Verilog HDL的基本概念，包括如何用其描述逻辑门和布尔代数方程，如何将不同功能的模块组成一个完整系统等。

第7章介绍了Verilog HDL的基本语法，重点阐述了组合逻辑和时序逻辑的描述方法。

第8章继续深入介绍了Verilog HDL，重点放在状态机的行为建模方面。通过几个实例阐述了使用硬件描述语言在行为模式下描述同步状态机的方法。

第9章专门介绍了异步（事件触发）状态机，从基本概念到设计应用都有详细的阐述。对于异步系统涉及的竞争冒险问题，也做了简要的讨论，并给出了解决方案。

第10章介绍了佩特里网络，以及如何用它实现时序和并行状态机。佩特里网络还可以用来控制同步信号引导多个并行状态机的操作。此外还介绍了如何使用D触发器来设计和综合佩特里网络。

每章都含有许多实例和解决方案，其中很多都被作者整合到实际运用的系统中。

Peter Minns BSc（H）PhD CEng MIET
Ian Elliott BSc（H）MPhil CEng MIET
Newcastle Upon Tyne

目　　录

第1章　有限状态机和状态图以及数字电路和系统设计的基本概念

1.1　概述

本书前3章的排版形式类似于连续的讲稿，属于系统化的学习资料。这样做的目的是便于读者可以很快地掌握同步状态机的基本概念，并可以运用T触发器和D触发器来构建自己的设计。其余的内容将逐步在后续的章节中展开，例如独热编码、异步状态机与佩特里网络等。

本章每一页讲稿虽然和前一页的内容是连贯的，但有时候读者会被引导到其他的章节，这取决于书中提出的问题。尽管如此，还是建议大家按照正常的顺序学习本书前3章的内容。

贯穿在章节中的一些“思考题”是用来测试读者对内容的理解和掌握程度。

为了方便大家理解“输入”和“输出”信号，本书中所有输入信号用小写字母表示，而输出信号则一律用大写字母表示。

在学习其他章节之前，请读者完成前3章的学习和作业。原因在于本书介绍的方法新颖而有效，如果使用得当，将能够快速地帮助大家构建基于状态机的数字系统。

本书第1~5章，第9章与第10章以逻辑门和布尔代数方程为基础，运用不同的方法设计状态机，读者可以掌握系统设计的每一个环节。

本书第6~8章是对Verilog硬件描述语言（HDL）的单独介绍。

1.2　学习资料

讲稿1.1

什么是有限状态机？有限状态机（FSM）是一种时序电路，通过控制一个或多个输入信号，实现电路在预设的各种状态里进行转换。每一个状态都可以看成是一个可以被状态机操作的稳固的实体。状态机可以从某一个状态转换到另一个状态，这种行为可以通过控制某一外界输入信号来完成。

图1.1展示了一个带有3种外部输入信号的FSM，分别是p、q以及时钟（Clock），FSM的对外输出信号分别是X、Y和Z。带有时钟的FSM一般被称作同

步 FSM，换句话说，那些不属于同步 FSM 范畴的可以被称作异步 FSM。然而，这里我们将主要讨论同步状态机，因它是带有时钟输入信号的。至于异步 FSM，将在后续的章节中讨论。

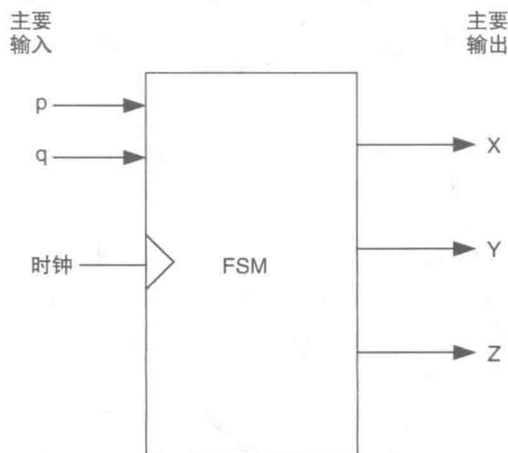


图 1.1　FSM 框图

正如前面所提到的，输入信号用小写字母表示，输出信号用大写字母表示。

只有当时钟脉冲出现的时候，同步 FSM 才能够在各种状态之间进行转换。

思考题

画出一个 FSM 框图，它带有 5 个输入，分别是 x、y、z、t 和一个时钟，以及 2 个输出 P 和 Q。

讲稿 1.2

在讲稿 1.1 的结尾，要求读者画一个 FSM，它带有 5 个输入和 2 个输出，答案如图 1.2 所示。

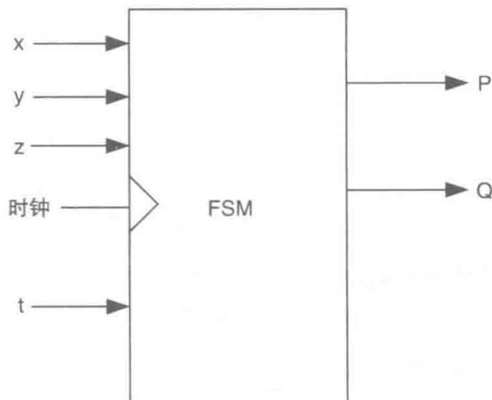


图 1.2　带有输入信号、输出信号和时钟输入信号的状态机框图

读者如果觉得自己画的和答案差别较大，可以重新学习讲稿 1.1。

FSM 的每一个状态都必须有清晰的定义。这是通过使用一定数量的内部触发器（对于 FSM 来说）来实现的。一个带有 4 个状态的 FSM 将需要两个触发器来构建，因为两个触发器可以产生 $2^2=4$ 个状态。每一个状态都有自己的状态编号，一般我们用 s0（第 0 个状态）、s1、s2 和 s3（以 4 个状态为例）来命名这些状态。

规则就是：状态的个数 $=2^{\text{触发器的数量}}$，而反之可以推断：触发器的数量 $=\dfrac{\lg(\text{状态的数量})}{\lg(2)}$。

因此一个有着 13 个状态的 FSM 将至少需要 4 个触发器（即 16 个状态，其中的 13 个状态是在 FSM 中被使用到的）来组成，计算公式如下：

$$\text{触发器数量}=\frac{\lg(13)}{\lg(2)}=3.7$$

结果必须四舍五入到整数，这里取 4。

思考题

1. 一个带有 34 个状态的 FSM 需要多少个触发器?
2. 如何给上述 FSM 中的各个状态标记名称?

讲稿 1.3

现在给出讲稿 1.2 中 2 个思考题的答案。

1. 一个带有 34 个状态的 FSM 需要多少个触发器?

答案是 $2^6=64$。

带有 6 个触发器的 FSM 可以容纳 34 个状态。一般来说 $2^4=16$ 个状态，$2^5=32$ 个状态，$2^6=64$ 个状态，$2^7=128$ 个状态等。

2. 如何给上述 FSM 中的各个状态标记名称?

答案是 s0，s1，s2，s3，s4，s5，s6，s7，…，s33。没有用到的状态是 s34 ~ s63。

除了用触发器来定义 FSM 的每个内部状态之外，定义 FSM 的输出信号是用组合逻辑电路来定义的。同时，每个触发器的输入端是通过组合电路和外部输入信号相连来驱动的。

讲稿 1.4

图 1.3 是米利（Mealy）FSM 的内部结构。

如图 1.3 所示，FSM 里有一定数量的输入信号连接到“下一状态解码单元”（组合逻辑电路）。由触发器组成的存储单元的输出信号被连接到输出解码电路后，再输出到外部。

触发器的输出同时被输入到“下一状态解码单元”，它们决定了 FSM 下一个状

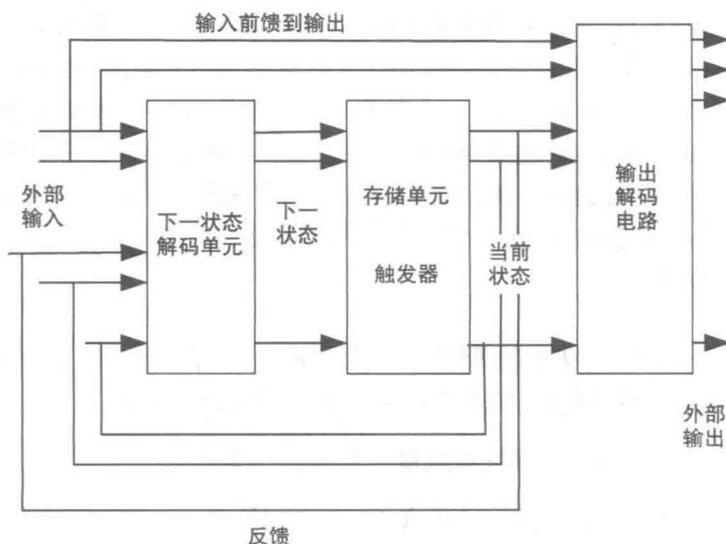


图 1.3 米利状态机的结构示意图

态的走向。一旦 FSM 进入到下一个状态，触发器会获取一个新的“当前状态”，这个“当前状态”将由“下一状态解码单元”产生。

外部输入信号会被直接输入到输出解码电路，这是米利 FSM 的主要特性。

讲稿 1.5

FSM 还有一种结构，被称为摩尔（Moore）FSM。

和米利 FSM 所不同的是，摩尔 FSM（见图 1.4）没有将输入信号直接输送到输出端。

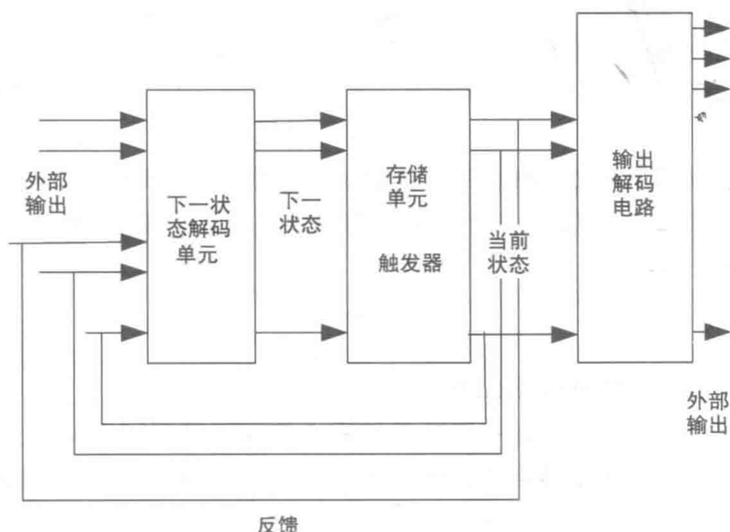


图 1.4 摩尔状态机的结构示意图

这种 FSM 十分普遍。需要注意的是图中外部输出只代表触发器的输出信号（不像米利 FSM，其外部输出包含了触发器的输出信号和一些外部输入信号）。

这两种 FSM 都将在后续做详细介绍。

讲稿 1.6

完成下面的填空：

摩尔 FSM 和米利 FSM 的区别在于它含有＿＿＿＿＿＿＿＿＿＿。

这意味着摩尔 FSM 的输出是取决于＿＿＿＿＿＿＿＿＿＿，而米利 FSM 的输出是取决于＿＿＿＿＿＿＿＿＿＿。

答案可以在讲稿 1.4 和讲稿 1.5 中找到。

讲稿 1.7

请大家再次观察摩尔 FSM 的结构图，如果将所有外部输入信号全部去除，只留下时钟信号。同时将输出解码电路移除，剩下的部分想必大家非常眼熟，如图 1.5 所示。

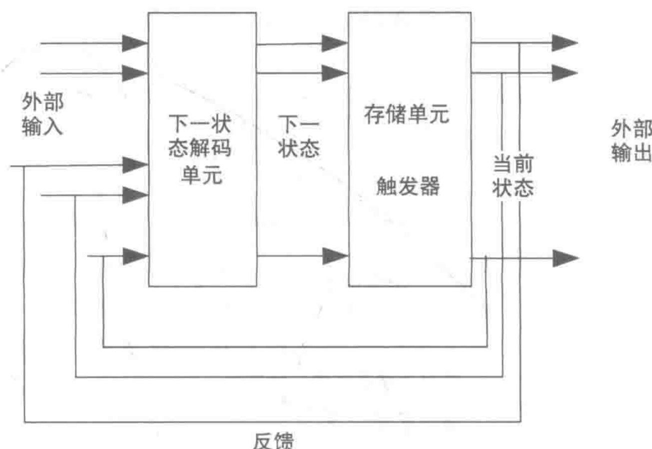


图 1.5　经典 C 状态机结构图

这种结构其实就是一个同步计数器，在许多场合都能够碰到。需要注意的是，如果需要计数器能够递增或递减来计数，则需要在外部额外增加输入信号，用来控制计数器的计数方向。

触发器的输出信号在这张图中是直接对外输出的。要注意的是，在同步 FSM 中（由时钟驱动），时钟是其中一个输入信号。

讲稿 1.8

起先有两种状态图，一种用来表示米利 FSM，另一种用来表示摩尔 FSM。这两种状态图分别叫作米利状态图和摩尔状态图。

如今，人们使用一种通用的状态图来表示这两种 FSM。本书也将使用这种状态图来表示所有类型的 FSM。

状态图包含 FSM 的每一个状态，并表示出每一个状态和其他状态之间的关系。通常用圆圈来表示一个状态（有些人喜欢用方块来表示），用箭头来表示状态的传输方向，如图 1.6 所示。

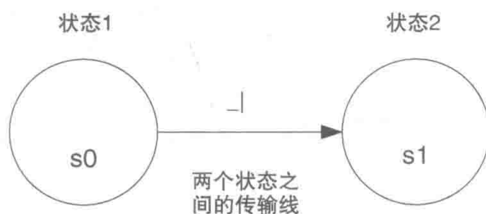


图 1.6 状态之间的传输示意图

除了状态之间的传输线，在传输线上方通常是输入信号的名称，如图 1.7 所示。

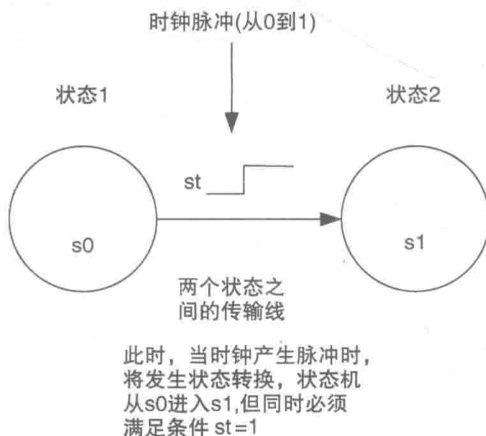


图 1.7 用于触发状态转换的外部输入信号

如图 1.7 所示，状态 s0 到 s1 转换的条件是外部输入信号 st = 1 以及输入时钟信号从 0 变为 1。

思考题

如果需要图 1.9 中的状态从 s0 转换到 s1，当满足 st = 0 时，还需要其他什么条件才能完成状态转换？

讲稿 1.9

讲稿 1.8 中思考题的答案如图 1.8 所示。

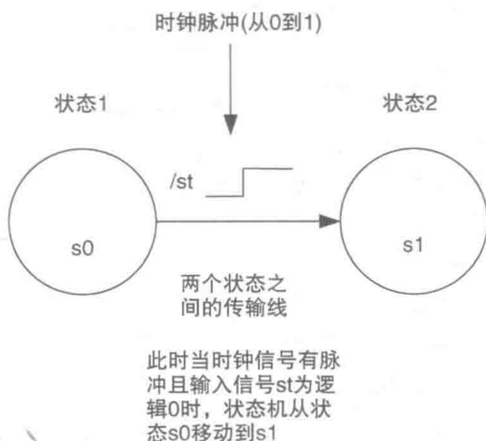


图 1.8　状态之间的外部输入信号

图 1.8 中，信号 st 已经被替换为/st，表示信号 st 的值必须为逻辑 0，才能触发状态从 s0 转到 s1，即 st 前面的斜杠表示“非”，因此当 st = 0，/st = 1。

需要注意的是，外部输入信号始终标记在传输线的上方。

状态图还需要表示出输出信号在不同的状态输出什么值。方法有两种：一种是在表示状态的圆圈（或方块）里标出输出信号的变化（或者值），如图 1.9 所示；另一种是在状态的圆圈（或方块）的旁边标出输出信号的变化（或者值）。

在图 1.9 中，输出信号 P 和 Q 是在状态圈的内部标出的。此刻，状态 s0 的情况下，P 的值为逻辑 1，当变换到状态 s1 时，P 的值为逻辑 0。输出 Q 在两个状态的转换过程中没有变化，始终为逻辑 0。

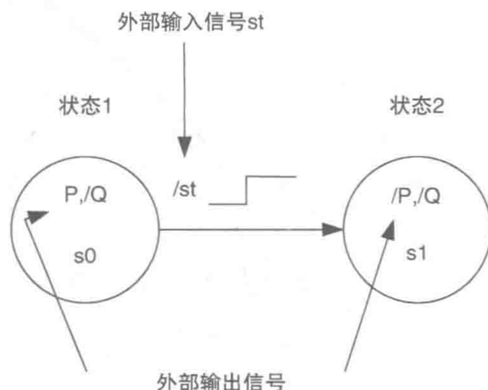


图 1.9　外部输出示例

图中像 st 这样的输入信号为主要输入信号，像 P 和 Q 这样的输出信号为主要输出信号。

思考题

根据图 1.9，画出含有输入和输出信号的 FSM 框图。

讲稿 1.10

FSM 框图如图 1.10 所示。

绘制类似的框图不难，因为输入信号处在传输线上方，而输出信号处在状态圈

内。它们的位置是固定的，且易于理解。

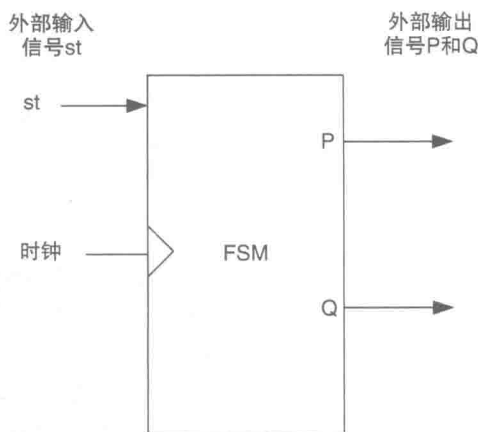


图 1.10　图 1.9 状态图所对应的 FSM 框图

回顾一下讲稿 1.2，其中提到每一个状态都对应一个特定的状态数，且需要一定数量的触发器来构成硬件电路。这些触发器是 FSM 内部构架的一部分，用来产生内部计数序列（它们实际上就像一个同步计数器，但是由外部输入信号控制）。触发器产生的内部计数序列是用来控制外部输出解码电路，这样输出信号的值可以随着状态的转换而变化。

讲稿 1.4 和讲稿 1.5 中所提到的米利和摩尔 FSM 中的存储单元就是由刚才所讨论的触发器组成的。

到这里，我们有必要分析一个简单的 FSM，看看它到底是怎么回事。这将用到前面所讲的所有知识，并引入一些新的概念。不过在进入下一讲之前，请读者先试着回答下面的问题。

思考题

1. 米利 FSM 和摩尔 FSM 的区别在于________。（答案在讲稿 1.4 和讲稿 1.5 中）
2. 状态图中的圆圈是用来表示________。（答案在讲稿 1.8 和讲稿 1.9 中）
3. 外部输入信号在状态图中的什么位置显现？（答案在讲稿 1.8 和讲稿 1.9 中）
4. 外部输出信号在状态图中的什么位置表示？（答案在讲稿 1.9 中）
5. FSM 里的触发器是用来________。（答案在讲稿 1.10 中）

讲稿 1.11　FSM 实例：带有指示功能的单脉冲发生器电路

设计思路是利用 FSM 设计能够在输出端 P 产生单脉冲的电路，产生脉冲的条件是只要当电路的输入信号 s 的状态为逻辑 1。在这种情况下，电路的另一个输出信号 L 将输出逻辑 1，但当 s 的值被释放时（状态变为逻辑 0），L 的状态也会被清零。输出信号 L 就好像一个拥有指示功能的存储单元，表明系统此时产生了一个脉冲。FSM 是用时钟驱动的，因此输入端有时钟输入信号。电路的框图如图 1.11 所示。

状态图如图 1.12 所示。

图 1.12 中，回旋箭头（就是从 s0 出发又回到 s0）表示当输入端 s 是逻辑 0（/s）时，FSM 会停留在 s0 状态，不管此时有多少时钟脉冲加载到 FSM 上。

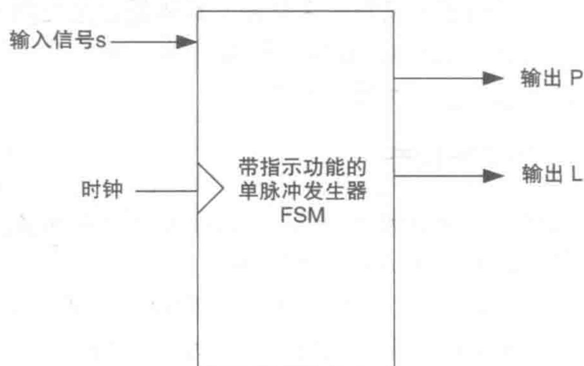


图 1.11　带有指示功能的单脉冲发生器 FSM 框图

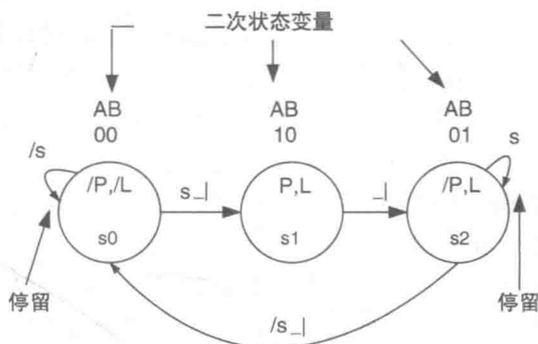


图 1.12　单脉冲发生器 FSM 状态图

只有当输入信号 s 的状态为逻辑 1（s），并在时钟脉冲到来的情况下，状态机才会从状态 s0 转变到状态 s1。一旦到达 s1，FSM 会将输出信号 P 和 L 的状态都更新为逻辑 1，并在下一个时钟周期到来时，从状态 s1 进入状态 s2。

FSM 在状态 s1 只停留一个时钟周期的原因是，当 FSM 到达 s1 的时候，从 s1 到 s2 的转变条件只有一个时钟脉冲的上升沿，就能立即触发，因此从 s1 到 s2 只需要一个时钟周期的时间。一旦 FSM 到达 s2，只要输入信号 s 的状态还是逻辑 1，FSM 就会停留在 s2；输入 s 的状态一旦变为逻辑 0（/s），FSM 就会在下一个时钟脉冲的上升沿到来时返回 s0。

由于 FSM 只在状态 s1 停留一个时钟周期，且 P = 1 只在状态 s1 存在，因此 FSM 会产生单个输出脉冲。注意到指示信号 L 会一直保持输出逻辑 1 直到 s 的状态被释放，提示用户一个脉冲已经产生。

注意在图 1.12 中，每个状态都有其独立的状态标识 s0，s1 和 s2。且每一个状态都由独立的触发器组合来构成：

状态 s0 用的是 A = 0，B = 0 的触发器组合，即两个触发器都被复位；

状态 s1 用的是 A = 1，B = 0 的触发器组合，即触发器 A 被激活；

状态 s2 用的是 A = 0，B = 1 的触发器组合，即触发器 A 被复位，B 被激活。

我们可以将这里的 A 和 B 触发器的状态称为“二次状态变量”。

触发器的输出也可以用于定义每一个状态，具体做法是将触发器的输出值用来表示状态机的每一个状态。图 1.12 中的 A 和 B 的编码序列是没有规律的，因为某些情况下两个触发器的状态会同时发生变化。

讲稿 1.12　输出信号状态

输出信号 P 的状态在何种情况下为逻辑 1，也可以用 A 和 B 的值来表示。例如在状态 s1，触发器的输出状态为 A = 1 和 B = 0。

因此，输出 P 的值可以表示为 P = A · /B（中间的实心点表示逻辑与操作）。这里需要再次强调的是触发器的作用是用来赋予每个状态一个独立的标识。

同理，输出 L 的值在 s1 和 s2 状态均为 1，因此 L = s1 + s2，L = s1 + s2 = A · /B + /A · B。

从这里可以推断出既然每一个状态都可以用触发器的输出来表示，那么状态机的外部输出也可以用触发器的输出来表示，这是因为外部输出本身和每个状态之间就存在着逻辑函数关系（P 在状态 s1 的值为逻辑 1，而状态 s1 可以用触发器输出表示为 A · /B），L 可以用 A · /B + /A · B 来定义。

给触发器的输出分配单独的赋值是一个任意的过程。理论上，只要每一个状态有一个独立的赋值组合，任意赋值均可以。这意味着一个状态机不能含有多于一个带有触发器输出格式为 A · /B 的状态。

实际上，一般在给触发器的输出赋值时，在每两个状态之间，尽量保持只有一个触发器的值发生变化。我们称之为“单位距离编码”[㊀]，上面的例子中没有采用这种编码方式，因为在 s1 和 s2 之间的转变中，两个触发器的值均发生了变化。

单脉冲发生器 FSM 的状态图也可以使用单位距离编码来完成，方法就是额外增加一个状态。我们将这个多出来的状态安插在 s2 和 s0 之间，输出和 s2 的 P 和 L 一样。

讲稿 1.13

图 1.13 给出了完整的使用单位距离编码表示的状态图。

新加的状态被命名为 s3，对应触发器的赋值为 A = 0 和 B = 1。对应输出 P = 0，和它在 s0 中的状态一样（因为当 s = 0 时状态机将回到 s0）。同时，L 的状态保持在逻辑 1 直到 s 被拉低，因为 L 是指示信号，在 s 被释放之前需要保持置 1。

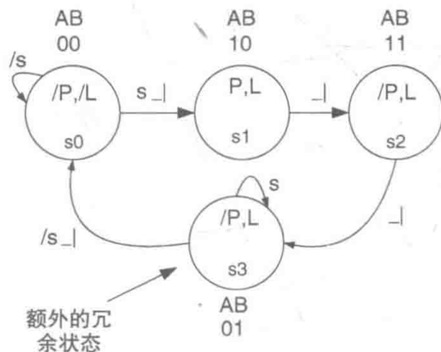


图 1.13　带指示功能的单脉冲发生器状态图

㊀ 这里的编码方式类似于格雷码的编码原理，相邻两个触发器的赋值组合里只有一位不同或者发生变化，例如 000，001，011，010 等。——译者注

在图中，新加的状态并没有让系统需要更多的触发器，因为两个触发器本身就可以容纳 2^2（=4）个状态（不熟悉这一层关系的读者可以复习讲稿 1.2 和讲稿 1.3）。

在这里，我们给系统加一个新的输入信号 r，当 r 的值被置高（逻辑 1）时，它将让系统的输出信号 P 以输入时钟的频率闪烁。当 r 的值被拉低（逻辑 0）后，系统将回到其原来的产生单脉冲的功能。

思考题

1. 画出上述描述的系统功能框图。
2. 画出新的 FSM 的状态图。

讲稿 1.14

新系统的功能框图如图 1.14 所示。

新的 FSM 的状态图如图 1.15 所示。

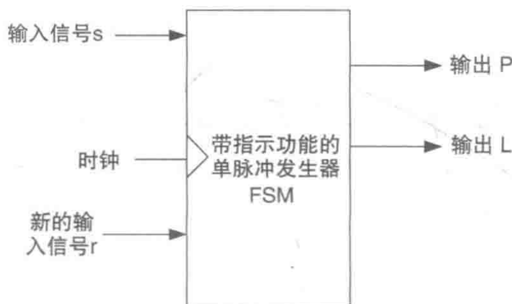


图 1.14　新系统 FSM 功能框图

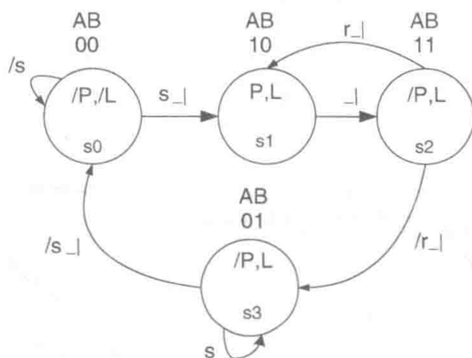


图 1.15　可以产生连续脉冲的单脉冲发生器

当加入新的输入信号后，系统多出了一个从 s2 回到 s1 的状态转换过程。注意，当 r = 1 时，FSM 在 s1 和 s2 之间随时钟的频率来回切换，直到 r = 0 时结束。

这种情况下，只要 r 的状态为逻辑 1，输出 P 的值将在 0 和 1 之间随着时钟频率来回不断地变化。

对于输出信号 L 不同的表达方法

图 1.15 中，L = s1 + s2 + s3 = A · /B + A · B + /A · B = A + /A · B。因此，L = A + B。如果对布尔代数公式不太熟悉，可以参考本书的附录 A。

除了上面的方法，还可以用取反的方法来表示 L：L = /(s0) = /(/A · /B)，这意味着当 A = 0，B = 0 的时候，L = 0。

低有效信号

图 1.16 给出了一个状态图的其中一部分，目的是为了显示低有效信号是如何

在系统中运作的，图中 s4，s5 和 s6 中的 CS 信号即为低有效信号。

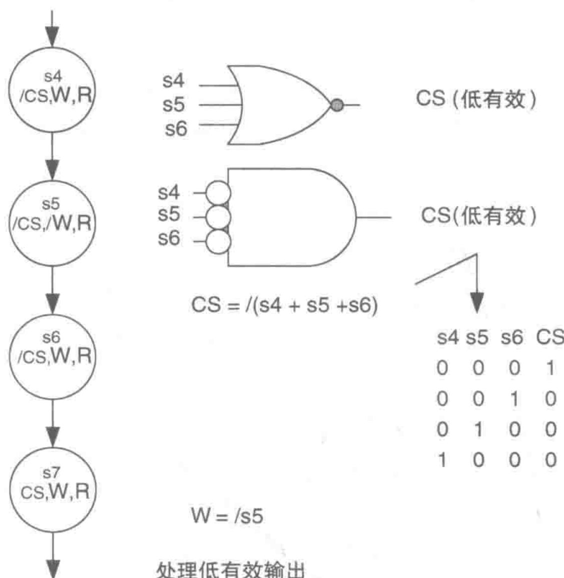


图 1.16　低有效信号

图中的 W 信号也是低有效信号。从图中可以推断出，要表示低有效信号，所有状态里输出为逻辑 0 的信号必须要取反。这在 FSM 里很常见，在以后会经常用到。

总之：

- 如果一个输出信号在所有状态里为逻辑 1 的情况多于逻辑 0 的情况，那么用低有效来表示它会比较简洁。
- 反之，如果一个输出信号在所有状态里输出逻辑 0 的情况多于逻辑 1 的情况，那么用高有效来表示会更加直观。

讲稿 1.15

在前几讲中，我们介绍了触发器输出信号的表达方式，这种表达方式通常我们称之为“二次状态变量”（见图 1.17）。

之所以称它们为“二次状态变量”，主要是因为它们（从 FSM 结构的角度看）属于 FSM 的内部参数。如果将外部的输入和输出认为是主要变量，那么把触发器的输出称为“二次状态变量”会显得更有意义，因为它们定义了 FSM 的状态。

FSM 的输出是依赖于二次状态变量与触发器存在的。回顾一下讲稿 1.5，会发现摩尔 FSM 的输出只取决于触发器的输出。之前介绍的单脉冲发生器的输出解码逻辑为 P = s1 = A · /B（参考讲稿 1.13）以及 L = s1 + s2 + s3 = A · /B + A · B +

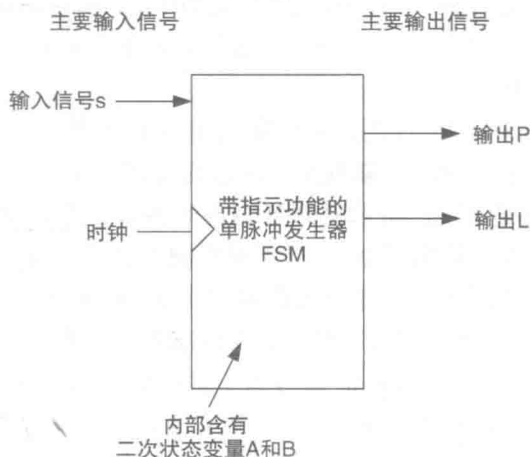


图 1.17　带有二次状态变量的 FSM 框图

/A · B = A + /A · B = A + B（辅助定律）。

也就是说，它含有一个与门和一个或门。意思是带有指示功能的单脉冲发生器是一个摩尔 FSM。

那么如何将它转换为米利 FSM 呢？

一种方法是使得输出 P 的值不但和状态 s1（A · /B）相关，而且还要跟时钟信号为逻辑 0 的时候相关。这样在状态 s1，P 输出的脉冲宽度和时钟信号为逻辑 0 的宽度一样。这就使得 FSM 拥有了一个从输入（时钟）到输出（P）的前馈路径。

思考题

画出上述的米利 FSM 的状态图。

讲稿 1.16

图 1.18 给出了讲稿 1.15 中思考题的答案。

注意到现在输出 P 只满足以下情况时才为逻辑 1：

- FSM 在状态 s1，且触发器输出 A = 1 以及 B = 0；
- 时钟必须为逻辑 0。

FSM 进入状态 s1 后，只有当时钟信号的状态为逻辑 0 时 P 才输出逻辑 1。具体过程为时钟信号从 0 变为逻辑 1，FSM 此时进入状态 s1，然后时钟信号变为逻

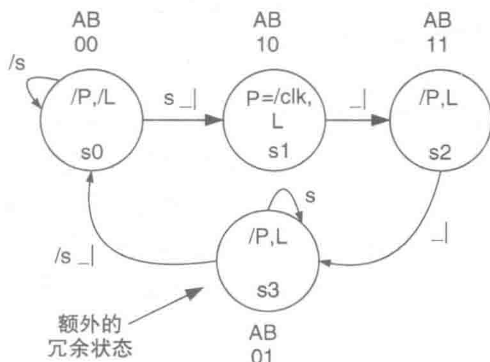


图 1.18　带有米利输出信号 P 的状态图

辑 0，此时 FSM 仍停留在 s1，而 P 的值由 0 变为 1。然后当时钟信号再次变为逻辑 1 的时候，FSM 会从 s1 进入到 s2，而此时触发器的输出将不再是 A · /B，因此输出 P 的值会回到逻辑 0。

图 1.19 给出了 FSM 的时序图，并将输出信号 P 在摩尔 FSM 和米利 FSM 的不同输出方式均显现出来，方便大家对照。通过对比可以看出，摩尔 FSM 在整个状态 s1 将 P 的值保持为逻辑 1，而米利 FSM 只在时钟信号为低的时候才将 P 置 1。

然而，在图的底部，给出了米利 FSM 里反向时钟（/clk）信号带有延迟的情况下，对输出 P 产生的影响。随着状态从 s0 变为 s1（/A · /B 变为 A · /B），由于时钟 clk 和其反向信号/clk 之间存在延迟，A 的状态发生变化之后，在输出 P 端出现了毛刺。从图中可以看出，当 clk 信号为 1 的时候，A 的状态跟随其产生变化，而/clk 信号在此时没有及时变为 0，因此产生了毛刺，这样的情况应该避免。

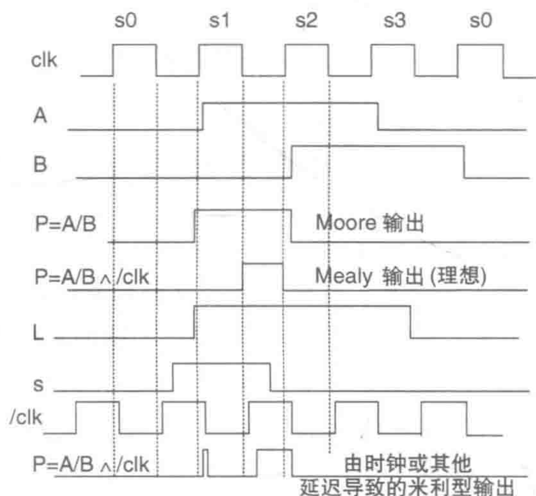


图 1.19 摩尔和米利 FSM 的时序图

毛刺出现的位置并不是特有的，不同信号的延迟可能在输出端 P 造成其他无法预估的结果（毛刺）。从根本上说，如果两个信号同时发生变化，则在输出端 P 会由于信号的延迟而产生一个毛刺（静电 1 竞争冒险）。

我们必须注意到在底部的输出信号 P，由于 A、B 和/clk 的延迟而随之产生了延迟。这样的延迟如果其时间间隔达不到一个时钟周期，可以忽略掉它，而且在实际操作中，通常是不会达到或者超过一个时钟周期的。

由此可以得出米利 FSM 的输出，最好不要用时钟来驱动产生。FSM 的参数变化尽可能使用单位距离编码，从而避免两个信号同时发生变化，这将在第 3 章进行详细讨论。

思考题

画出一个 FSM 的状态图，要求当输入信号 m 变为逻辑 1 时，FSM 的输出信号

在接下来的 3 个状态分别输出“101”，并且 m 的值需要在下一次 FSM 输出“101”到来之前清零。

讲稿 1.17

解决讲稿 1.16 中思考题的办法就是利用单脉冲发生器的状态图，向里面加入更多的状态，产生一个输出“101”的状态序列。图 1.20 给出了整个设计是如何通过一步一步地加入新的状态来完成的。

一开始等待输入信号 s 变为逻辑 1。因此在状态 s0 所做的事情就是等待 s = 1。一旦输入 s = 1 且时钟从 0 变为 1（上升沿）出现，FSM 将进入下一个状态 s1，输出信号 P 将变为逻辑 1。

下一个状态 s2，输出信号 P 变为逻辑 0。再下一个状态 s3，输出信号 P 变为逻辑 1。

值得注意的是在状态 s3，FSM 在时钟上升沿到来时就要离开这个状态，因此输出信号 P 保持逻辑 1 的时间只能等同于时钟信号的一个脉冲宽度。

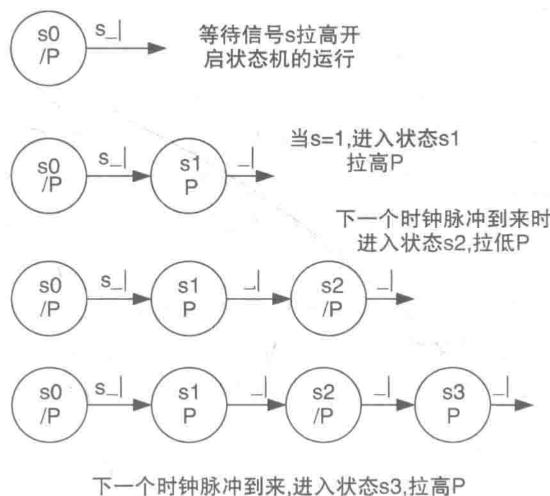


图 1.20　产生“101”输出序列的过程

最后一个状态所要做的事情就是等待输入信号 s 变为逻辑 0。这样 FSM 就能够返回状态 s0。

思考题

完成上述 FSM 的状态图。

讲稿 1.18

完整的状态图如图 1.21 所示。

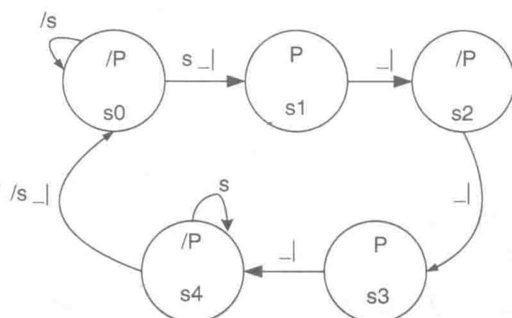


图 1.21 完整的“101”输出序列状态图

输出信号 P 的布尔代数表达式为 P = s1 + s3。然而，也可以将 P 改为米利型输出，这里可以加入一个限制条件，输入信号在 y = 1 的情况下，在状态 s1 和 s3 输出信号 P 才为逻辑 1。表达式为 P = s1 · y + s3 · y，这样 P 虽然必须满足条件在 s1 和 s3 为逻辑 1，但是现在有了附加条件 y 为逻辑 1。

关于状态的停留

当输入信号的值不满足状态机进行状态切换时，我们用停留箭头来表示状态机维持当前的状态，并且将维持现状的输入信号值也标记在停留弧线的上方。其实在设计 FSM 时它们不是必须出现的，因为停留在某个状态并不影响系统生成电路并运行状态机。事实上，它们只是起到了方便读者理解的作用，让大家更加易于读懂设计的意图。因此从今往后，状态停留只会在增进易读性的地方出现。

思考题

现在请大家尝试将状态图改为一个基于时钟信号输出“1010”序列的状态机（就像图 1.21 中那样），但是输出 P 的值在状态 s3 会受到一个新的输入 x 的限制，当 x = 0，FSM 在 P 上产生的序列为“1000”，当 x = 1 时，状态机的输出才为“1010”。

讲稿 1.19

讲稿 1.18 中思考题的状态图如图 1.22 所示。

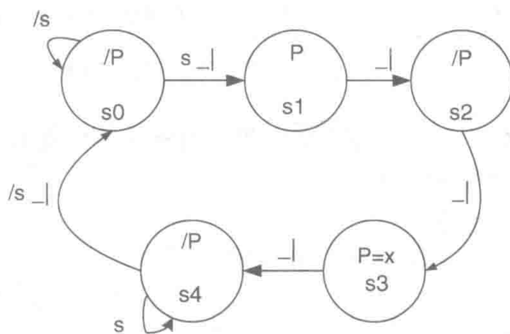


图 1.22 将输出 P 改为米利型输出的状态图

图中的输入信号 x 用来在状态 s3 判断输出 P 的值，在状态 s3，只有当 x 的值恰巧为逻辑 1 时，P 才在时钟上升沿到来后输出逻辑 1，即一个脉冲。而在状态 s1，P 会始终输出脉冲。当输入 x = 0 的情况下，在输入

信号 s = 1 的条件满足后，P 会输出一个“1000”的序列。而在 x = 1 时，P 的输出序列为“1010”。

这是个很典型的米利 FSM 的模型，因为输出 P 受到了状态机和输入信号 x 的共同影响，即输入 x 被前馈到输出解码逻辑中，所以 P 的表达式可以写成：P = s1 + s3 · x。

如果想把输出 P 的序列改为当 x = 1 时输出“1000”，当 x = 0 时输出“1010”也不困难。

思考题

1. 将满足上述修改方案的 P 的布尔代数式写出来。

2. 使用单位距离编码方式来表达状态图（参考讲稿 1.12 和讲稿 1.13，弄清楚这样做的原因）。

3. 画出时序图。

讲稿 1.20

讲稿 1.19 的思考题中，第 1 题布尔代数式如下：P = s1 + s3 · /x。

当 x = 0 时，P 输出序列“1010”。需要注意的是，此时影响 P 的变量是/x，而不是 x。

第 2 题使用单位距离编码方式来绘制的状态图如图 1.23 所示。

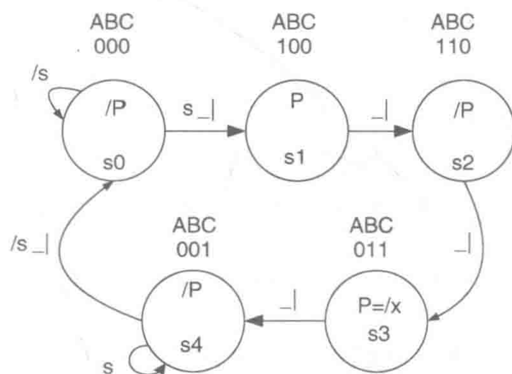


图 1.23　使用单位距离编码的状态图

在状态 S3 圆圈里，公式 P 成立的条件是输入信号 x 的值为逻辑 0（如果状态圆圈里的空间不够大，公式也可以写在圆圈的外面）。对于图中的二次状态变量，可能不同的人得出的结果会不一样，这并不重要，因为只要不是为了使用单位距离编码，二次状态变量的赋值其实没有一个所谓的“最佳答案”的组合。

但我们会发现，图 1.23 中从状态 s2 到 s3 并不是单位距离编码模式，触发器 A 和 C 在状态变化时同时发生了值的改变。那么如果需要遵循单位距离编码模式，

就需要在 s2 和 s3 之间加入一个冗余的状态（和在讲稿 1.13 中单脉冲发生器的情况一样）。

然而，添加冗余状态时必须谨慎。如果冗余状态加在 s1 和 s2 之间，会影响到 P 的最终输出序列，如果我们要的是“1010”，但是最终输出却是“10010”。合适放置冗余状态的位置是 s3 和 s4 之间，或者 s4 和 s0 之间，因为这两个位置不处于 P 的输出序列里，不会影响到最终结果。

在讲稿 1.21 中我们会讨论时序图。

讲稿 1.21

讲稿 1.19 中思考题第 3 题的答案如图 1.24 所示，这里的答案是基于图 1.23 中的二次状态变量来画的，如果用了不同的单位距离编码组合，答案会有所不同。

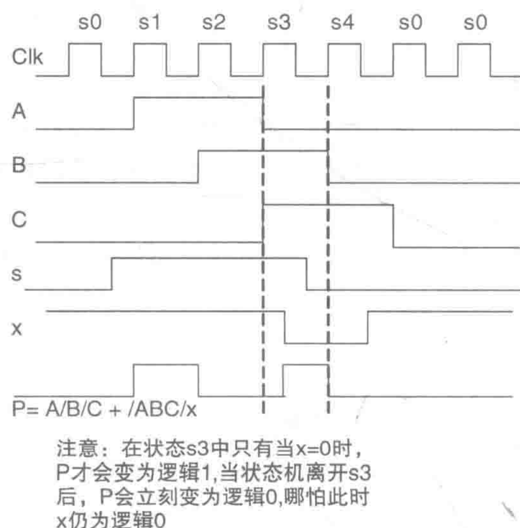


图 1.24 输入信号 x 作用于输出信号 P 的时序图

注意到图中的输入信号 x 在状态 s3 的时钟脉冲内发生了变化，从 1 变为了 0。这种变化会给 P 的输出带来影响，注意输出信号 P 在 s3 的时钟周期里不是全为 1。这种情况比较符合实际，因为外部输入 x（实际上任何外部输入）可以在任何时刻发生变化。

1.3 小结

至此，关于 FSM 的基本知识以及如何绘制状态图已经全部讲完了。其中包括：

- FSM 的输出是如何受到二次状态变量的影响；
- 二次状态变量是随机赋值的，但是遵循单位距离编码的方式较为实用；

- 如何构建米利或者摩尔 FSM，它们的输出信号是如何用公式来表达的。

然而，必须认识到状态图其实是由逻辑门和触发器等单元组成的电路。如何构建电路是比较程序化的举动，这将在第 3 章中具体讨论。

在第 2 章中我们会学习几个 FSM 的例子，学习如何使用它们控制外部器件或者电路，并绘制状态图。后面的章节节奏会加快，原因在于我们默认大家对之前所讲的内容理解较为透彻。

第 2 章　使用状态图控制外部硬件分系统

2.1　概述

系统设计过程中，经常需要外接一些模块，例如计时器/计数器、模 - 数转换器（ADC）、存储单元等，并使用握手信号来与这些器件进行通信。

本章将介绍如何运用状态图（即 FSM）来控制这些器件（模块）。这不但扩展了 FSM 的应用范围，而且可以为设计师在相对较短的时间内提供硬件解决方案。

在后面的章节中，本章介绍的一些模块控制方案将有机会得到进一步运用。

2.2　学习资料

讲稿 2.1

FSM 设计中最常见的一个现象就是系统进入某个状态之后，需要等待一段预设的时间。例如，状态机需要将特定的信号对外输出一段时间，然后再停止输出。此时可以在系统里增加一定数量的连续的冗余状态，并让这个特定的信号在这些增加的冗余状态中保持输出所需的固定的值，这种做法很浪费（不但增加了不必要的状态数量，而且增加了触发器的数量），但系统的延迟很短。一种更好的方法是使用一个外部计时器单元，让 FSM 对其进行控制。

典型的计时器如图 2.1 所示。

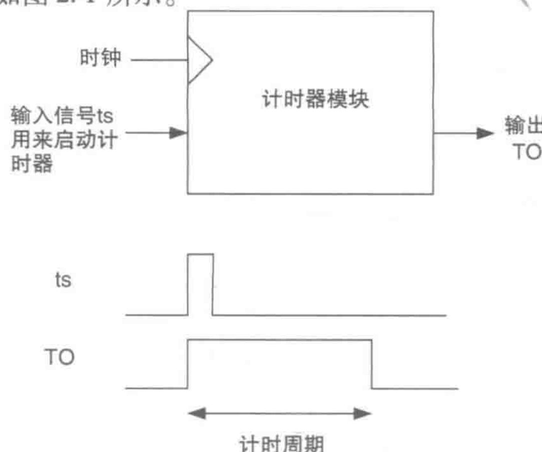


图 2.1　计时器模块

图中计时器有两个输入，一个时钟输入信号 clk 和一个开始计时的启动信号 ts，还有一个输出信号 TO。从图 2.1 中的时序图可以看出，当计时启动信号 ts 产生一个脉冲时，输出信号 TO 将被置高。TO 将在计时结束后被释放，回到低电平状态。

图 2.2 中，FSM 输出 TS 信号（状态 sn +1）用来启动计时器。然后在下一个时钟脉冲到来时，FSM 进入计时状态（状态 sn +2）。在计时状态里，TS 信号回到逻辑 0，计时结束信号“to”处于状态机的监控之下，一旦“to”回到低位，标志着计时结束。这时，状态机会离开计时状态。

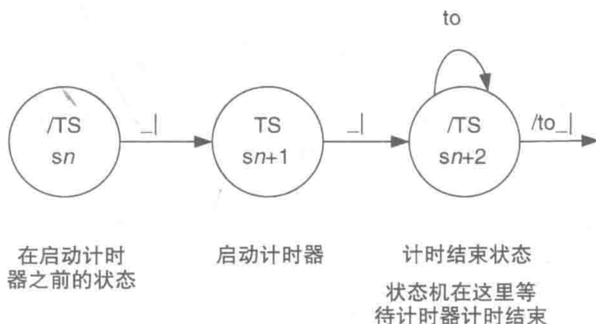


图 2.2　控制计时的状态序列

有必要强调一下，FSM 会在状态 sn +2 停留，直到信号“to”变为低。因此，在 sn +2 状态停留的时间由计时器决定。

同时还要注意，这里信号“to”的字母是小写，这和图 2.1 是不同的，因为对于状态机来说，它是输入信号，而信号“TS”用了大写字母，因为它是从 FSM 输出的信号。

讲稿 2.2

现在请大家看另一个运用计时器的例子。

图 2.3 中，FSM 在控制一个计时器模块。这里的计时器和讲稿 2.1 中提到的计时器有所不同，它没有时钟输入。事实上这是一个基于 *RC* 充电电路的计时器（例如市面上可以买到的 555 计时器模块）。但是只要是基于 FSM 的应用，其实质性和之前相比并没有发生变化。如果我们使用的是 555 计时器的话，这里实际的时延可以表达为 $TO = 1.1 \times C \times R$。

FSM 根据信号 st 进入状态 s0。这里要做的就是将输出信号 *P* 拉高一段时间（这个时间段由 *RC* 电路的时间常数来控制），然后再复位。

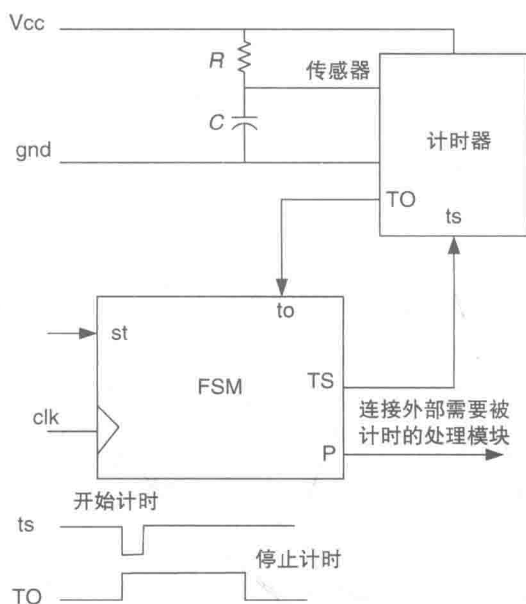


图 2.3　运用计时模块的系统框图

思考题

画出上述状态图。

讲稿 2.3

图 2.3 所描绘的设计思路对应的状态图如图 2.4 所示。

输出信号 *P* 在状态 s1 中被置 1，由于计时器也在此状态中被启动，在 FSM 进入计时状态 s2 之前，*P* 会一直保持高电平。FSM 会在状态 s2 中等待信号“to”回到逻辑 0，然后进入状态 s3，将输出信号 *P* 拉低。

这种设计方法可以让 FSM 在状态 s2 停留的时间可控，等多久只取决于 *RC* 电路的时间常数。

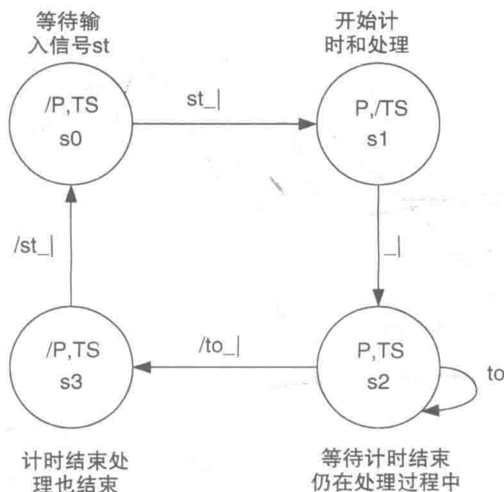


图 2.4　运用计时器模块的状态图

讲稿 2.4

在看完如何使用状态机控制类似计时器这样的模块之后，下一步可以开始学习使用状态机来控制其他类型的外部器件。这其实就是 FSM 的主要作用——控制各

种外部器件（电路）。

现在开始学习如何使用 FSM 控制模 - 数转换器（ADC）和存储单元。

控制模 - 数转换器（ADC）

模 - 数转换器是将模拟信号转换为数字信号的器件。它们使得数字电路（处理芯片）能够处理从真实世界输入的信号（自然界的模拟信号）。大部分含有 ADC 模块的系统都带有微处理器（例如单片机）芯片。然而，很多时候会看到系统（或系统的一部分）使用定制的芯片，即可编程逻辑器件（PLD），或者现场可编程门阵列（FPGA）。请大家看图 2.5。

图 2.5 中的 ADC 模块带有输入信号 SC（开始转换）和输出信号 eoc（结束转换）。

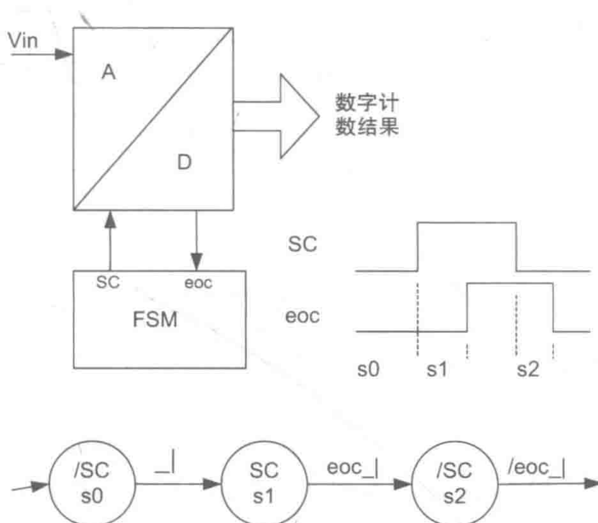


图 2.5　用状态图来控制一个 ADC 模块

ADC 模块的模拟输入（Vin）和数字信号输出与外部电路相连，并不受 FSM 的控制，因为它们参与构成系统的数据流部分。FSM 在这里的作用是控制系统的各个模块（这里指的是 ADC）。

图 2.5 中的状态图给出了如何控制 ADC 的过程。

FSM 在状态 s1 启动 ADC 进行模 - 数转换，并等待 ADC 的 eoc[㊀] 信号从 0 变为 1，表示 ADC 的输出端有完成转换的数字信号输出。此刻，FSM 进入 s2，并等待 eoc 信号从 1 变为 0 之后再进入后面的状态。

现在请看图 2.6，一个小型数据采集系统（DAS）。

㊀ 注意某些 ADC 器件用一个"忙碌"（busy）信号代替"eoc"信号。当转换开始信号 SC 被置高时，busy 信号也被拉高，当转换结束时 busy 信号被拉低。

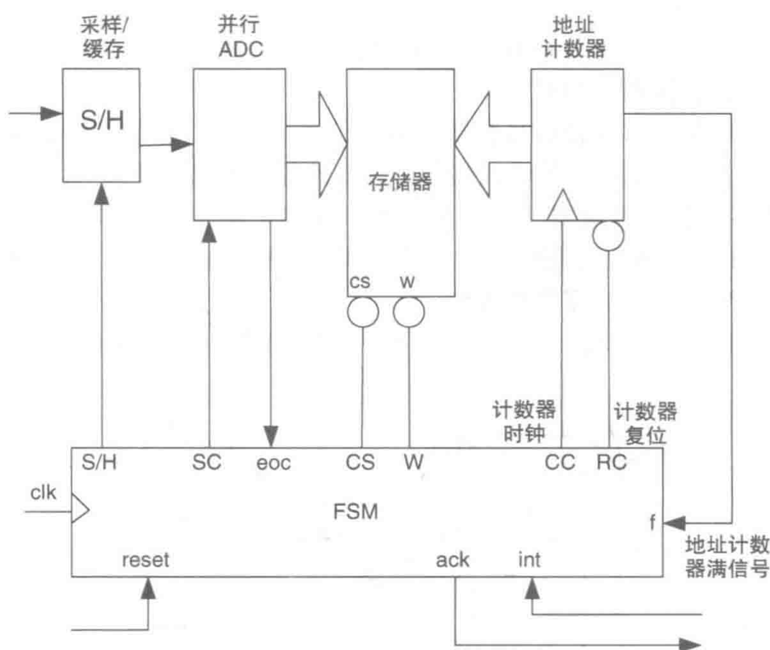


图 2.6 小型数据采集系统框图

在这个系统里有一个 ADC 以及一些其他的外部器件。

讲稿 2.5

图 2.6 中的系统和之前接触到的例子相比比较复杂；然而，它可以被分解为好几个部分，方便大家理解。

系统利用状态机分别控制一个“采样和保持”（S/H）模块、一个 ADC 模块、一个随机存取存储器（RAM）以及一个简单的二进制计数器。

所有这些外部器件和 FSM 组成的系统可以完成以下两个功能：

- 对外部的模拟信号进行采样；
- 将数据存储到 RAM 里。

从硬件角度看，这样的系统通常是一个便携式的设备。

在设计状态图之前，需要事先讨论 FSM 如何控制 RAM 和计数器。

如图 2.7 所示，一个被 FSM 控制的存储芯片。

在状态 $sn+3$ 里，系统在读信号或者写信号的上升沿到来时进行存储芯片的读写操作。注意到存储芯片有地址输入端口（通常叫做地址总线）和数据端口（通常叫做数据总线）。如果系统只能从存储芯片读数据，那么数据总线只能作为输出。如果存储器件是一个随机存取存储器（RAM），那么数据总线是双向的。这意味着/R 和/W 信号可以用来定义数据总线是作为输入（当/W 有效）还是作为输出

(当/R 有效)。此外，还有一个片选信号作为存储芯片的输入，表示系统在进行读写操作之前必须先选中所控制的存储芯片。

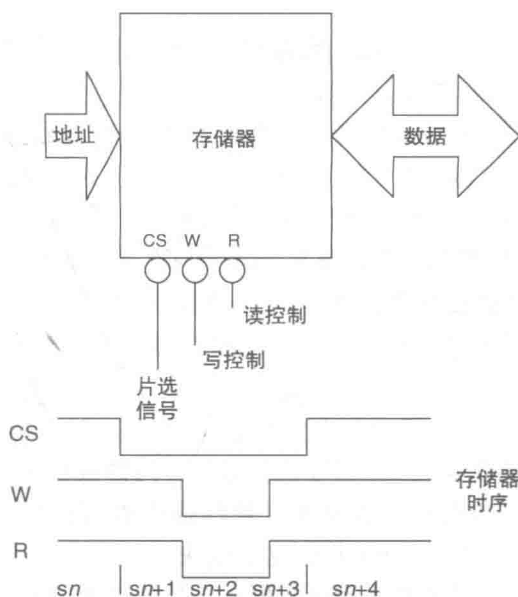


图 2.7 控制一个存储器件

关于存储器件的时序关系

图 2.8 给出了和存储器件相关的时序关系。

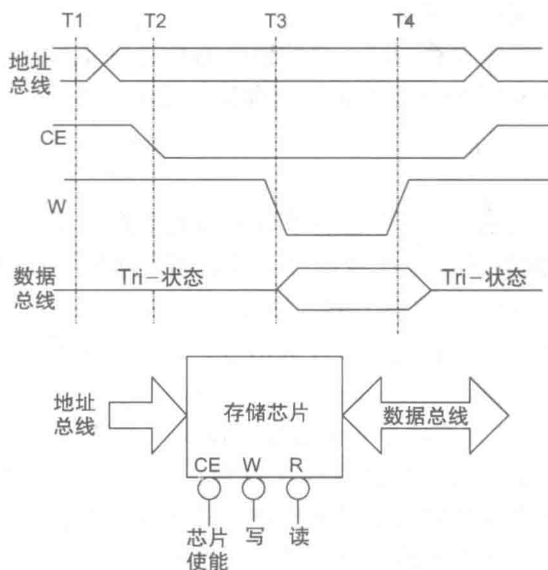


图 2.8 控制存储器件的时序图

地址总线在时间 T1 选择特定的地址空间（地址总线的值在 T1 之后开始发生变化）。随后片选信号 CE 在时间 T2 有效（低有效）。图中考虑到了传输线上的延迟。

在时间 T3，写信号 W 被激活（低有效），存储芯片的数据总线的端口性质由原来的三态变为（数据）输入。

（经过一段时间的写数据操作之后）在时间 T4，写信号 W 被拉高，然后片选信号 CE 也会被拉高。片选信号被拉高后，存储芯片不再被系统选中，即不再接受任何数据的输入。在写信号 W 从 0 变为 1 的过程中，数据总线上的内容将被全部写入存储芯片。有一些存储芯片会将 W 信号和 CE 信号同时置高。但在这里，系统会将 CE 信号保持足够的有效（逻辑 0）时间，让 W 信号控制总线完成写操作。通常地址解码逻辑电路会让 CE 信号上的传输延迟比 W 信号更长，因此这里两个信号没有同时发生变化。

在一个由 FSM 控制的系统中，上述过程可以通过图 2.7 里的波形图来实现。另一种可能实现的方法就是让 CE 信号在存储芯片内部被延迟。这种方法实现的途径是利用 FGPA 芯片里的资源，用 HDL（硬件描述语言）编写一个存储模块。当然在 FPGA 芯片里也可以用 FSM 来控制这个模块。

延迟 CE 信号的主要目的在于确保数据在芯片被选中的时段内将所有数据写到 RAM 中去。

注意：芯片的 CE 和 W 信号应该用 FSM 来控制，无论是向芯片里写数据还是从芯片里读数据。

如果写（W）信号被读（R）信号替代，那么存储芯片就变成了只读芯片，存储于芯片中的数据只能向外输出。

读的过程和写的顺序大致相同，之前讨论的片选信号上的延迟同样适用。

下一讲将向大家介绍 FSM 是如何控制存储芯片的。

讲稿 2.6

要访问存储器件（芯片），片选信号必须被激活（这里的片选信号是低有效，即必须为逻辑 0）。然后通过拉低写信号，加载需要被写入的数据。一段时间后，将写信号拉高（逻辑 1），将总线上的数据写入 RAM。

要读取芯片里的内容，首先激活片选信号，即将其拉低，延迟一小段时间后，将读信号拉低。

在大部分情况下，“片选信号和读信号”以及“片选信号和写信号”这两组控制信号可以同时被置高。通常也是在这个时候，存储芯片完成读和写的工作。但是，如果对读和写所需要的时间不是很确定，最好将片选信号的状态维持逻辑 0 足够长的时间，或者直接先将读或写信号拉高，然后再拉高片选信号。

实际操作过程中，数据总线在系统的控制下会保持几 ns 左右（一般 10ns）的

有效时间，让数据写到存储器中或将数据从存储器中读出来。但是在基于 FSM 的系统中，设计人员需要通过添加一个新的状态或者在片选信号上加上一定的延迟来确保数据读写的正常和稳定。

图 2.7 中的时序图很好地解释了上面所说的内容。

当从存储器件读数据或者向里面写数据时，这两个动作是从控制存储器件的处理器或者状态机的角度来看的。在带有微处理器的系统里，控制系统的器件是微处理器。而在这里，控制系统的是 FSM。

思考题

试着画一个控制存储器件写数据的状态图。

讲稿 2.7

前两讲所阐述的控制存储器件的状态图如图 2.9 所示。

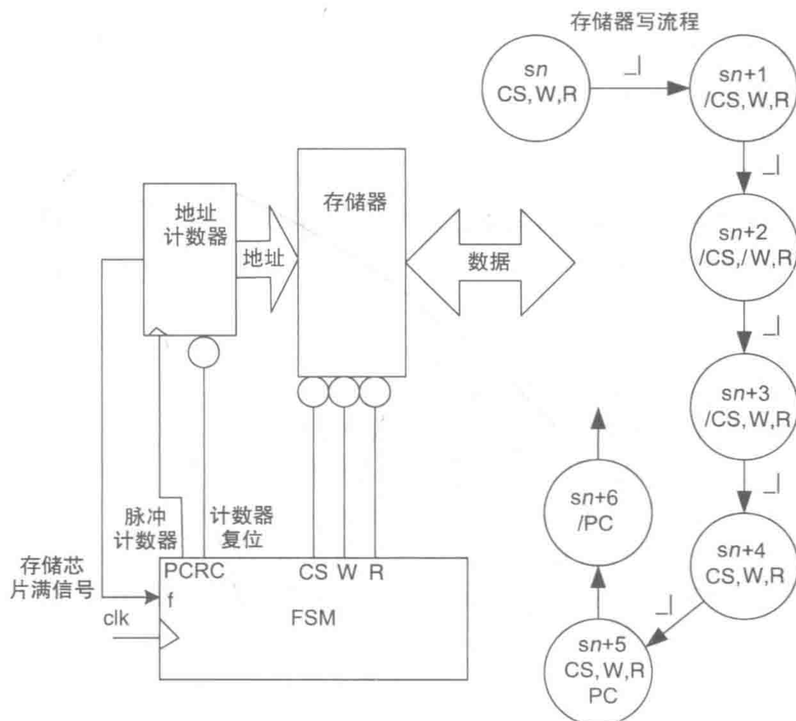


图 2.9　使用 FSM 来控制存储器件的写操作

在状态 sn，所有控制信号都是处于未激活状态。在状态 sn + 1，片选信号（CS）首先被激活（低有效）；然后在状态 sn + 2，写信号被激活。在状态 sn + 3，写信号被释放，此刻系统将数据写到存储器中。最后，在状态 sn + 4，片选信号（CS）被拉高，释放存储器件。

数据交换一般在地址总线访问的那个存储单元发生，无论是读还是写。要访问另外的存储单元，需要选择新的地址，这里的地址选择是由地址计数器来完成的。这也是图 2.6 以及图 2.9 中的计数器的用处所在。这样，随着二进制计数器值的逐步递增，系统可以按照顺序访问每一个存储单元。

在状态 $sn+5$，信号 PC 被拉高。这代表计数器开始工作，系统将随着计数器值的递增，访问下一个地址空间。

信号 RC 为低电平可以将计数器清零。计数器随着 FSM 每一次向信号 PC 发送脉冲而递增。这样每一个存储单元对应的地址都可以被依次访问到。要注意的是，在启动地址计数器（转到下一个地址空间）之前，片选信号必须被释放。因为当芯片被选中时，此时对应的地址是固定的，只有在完成了当前所对应的地址空间的读或者写后，才能对下一个地址空间进行操作。

讲稿 2.8

思考题

掌握如何控制单个外部器件后，请读者尝试用 FSM 来控制图 2.6 里的整个系统，并绘制出状态图。

系统功能描述如下：

在信号 int 端收到中断信号之前，FSM 会一直处于状态 s0。收到中断信号后，FSM 将做如下事情：

- 对数据采样；
- 进行模－数转换；
- 将转换结果保存到存储器中；
- 将地址计数器累加并指向下一个可用的存储空间。

FSM 必须反复不停地完成上述过程直到存储器存满为止。当计数器上的信号 f 为逻辑 1 时，即代表存储器已经存满。

这时候，FSM 将通过信号 ACK 发送一个响应信号给外部便携终端；随后当信号 int 为低时（注意，信号 int 一开始的时候是被置高的），FSM 就需要返回状态 s0，为下一次数据采集做好准备。

讲稿 2.9

讲稿 2.8 中的思考题要求绘制的状态图如图 2.10 所示。可能有些读者绘制的状态图和我们提供的不一样，解题的思路可以多种多样，这种题目没有“唯一”模板。不过在图 2.10 中给出的参考答案十分简洁直观。

首先要明确的是存储器件的地址计数器所对应的复位信号 RC 在状态 s0 是被激活的（低有效）。在后面的状态中，它被一直拉高（置 1）。除了状态 s0 和 s1，信号 RC 的值并没有在其他状态里体现，不过后面系统的一系列操作暗示了它会被

一直拉高。

当输入信号 int 有效时，FSM 进入到状态 s1，释放地址计数器上的复位信号，并同时将采样和保持放大器激活，即 S/H = 1。

当下一个时钟周期的脉冲（上升沿）到来时，FSM 进入状态 s2，此时 S/H 信号仍然有效，且模 - 数转换器的转换信号 SC 被激活，即 SC = 1。FSM 开始等待转换结束信号 eoc = 1；然后，下一个时钟脉冲（上升沿）到来时，FSM 进入状态 s3，并在此状态中将 S/H 信号释放（S/H = 0），因为此时模 - 数转换器已经完成了将模拟信号转换为数字信号的工作。在状态 s3，还有要做的是激活片选信号 CS（CS = 0），并等待信号 eoc 回到低位（逻辑 0）。当这些条件都满足以后，在下一个时钟上升沿到来时，FSM 进入状态 s4，存储器写信号 W 被激活（W = 0），并将存储器件的数据总线的性质设为输入。这样 ADC 的数字输出将变为存储器件的输入。

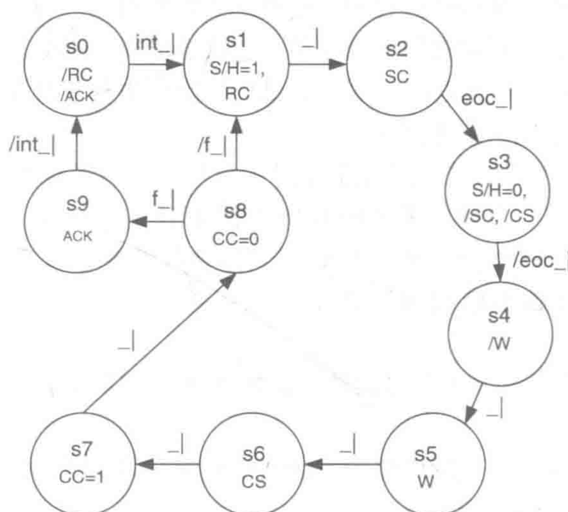


图 2.10　数据采集系统（DAS）的状态图

再下一个时钟上升沿到来时，FSM 进入状态 s5，此时 W 信号被释放，并将 ADC 的结果写入存储芯片。

在下一个状态 s6，FSM 会释放存储芯片（CS = 1）。

讲稿 2.10

此时的状态机处于状态 s6，即将进入状态 s7，在 s7 里信号 CC 将被激活。下一个时钟脉冲到来时，FSM 进入状态 s8，信号 CC 被拉低。信号 CC 的值从 0 到 1 的转变过程，也是地址计数器加 1（递增）的过程。注意到在状态 s6，CS 和 W 信号已经分别被拉高（释放）。

在状态 s8，FSM 有两个选择，要么进入 s1，条件是当信号 f 的值是逻辑 0，即

重复状态 s1 到 s8 的循环；要么是进入状态 s9，此时信号 f 的值为逻辑 1。

信号 f 的作用是用来指示地址计数器是否到达其最大计数值，即存储芯片的最后一个地址空间。如果系统还没有访问到存储芯片的最后一个地址空间，说明存储芯片还未存满，即 f=0，则再一次进入另一个循环。否则 FSM 将进入 s9，且 ACK 信号被激活（拉高），让外部器件得知 FSM 已经完成工作。此时 FSM 会等待 int 信号被拉低以便回到状态 s0。

信号 int 被拉低以后，FSM 开始等待下一个操作循环，首先需要得到的是信号 int 从低到高的转变。必须注意的是，外部器件需要将信号 int 拉低来完成握手的过程。当看到 int 信号被拉低后，FSM 将 ACK 信号拉低，表示状态机和外部器件的握手通信完成。

数据采集系统（DAS）是一个比较复杂的系统级应用，体现了如何运用状态机控制一系列操作流程，从而达到控制一些外部器件的目的。

从中可以看出，运用状态图来表述整个过程会显得很直观。然而，读者也可以通过使用本书介绍的一些方法设计属于自己的状态图去控制其他外部器件。

现在总结一下已经学到的方法：

- 将时钟信号和其他输入信号连接到一个与门，并连接到状态机的外部输出端，形成米利型输出（具体参考讲稿 1.16 的图 1.18）；
- 使用冗余状态构成单位距离编码（具体参考讲稿 1.12 和讲稿 1.13）；
- 控制外部计数器模块可以在某个状态停留一段预设的时间（参考讲稿 2.1）；
- 使用 FSM 来控制其他外部器件，例如 ADC（讲稿 2.4）和存储器件（讲稿 2.5）。

如何将状态图演变为实际电路的方法和步骤将在第 3 章详细阐述，这里还有一些其他的技巧需要事先介绍一下。

讲稿 2.11

请大家看一下图 2.11 所示的 FSM 框图。

这个 FSM 有一个特性，当输入信号 d 被连续拉高两次的情况下，信号 P 才会输出一个时钟脉冲，并且 FSM 需要时钟来驱动。

思考题

试着画出图 2.11 所示 FSM 的状态图，然后到讲稿 2.12 中找答案。

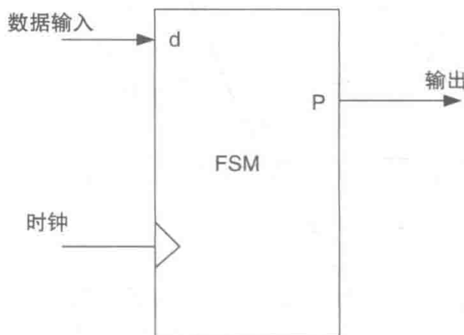


图 2.11 FSM 框图

讲稿 2.12

现在将讲稿 2.11 中的框图重复一遍。

这里的要点是观察输入端的信号值的变化。也就是说要观察到信号 d 被拉高两次。

这是关键，因为 FSM 需要判断什么时候 d 被激活了两次。要达到这个目的，需要先观察到 d 从低变高，然后变低（此刻 d 已经变高变低一次了）。继续观察 d 再次变高，然后 d 再次变低（此刻 d 已经变高变低两次了）。

状态图如图 2.13 所示。

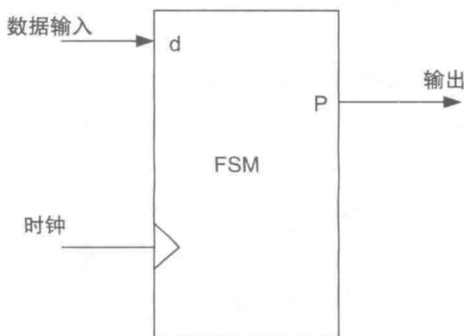


图 2.12　FSM 框图

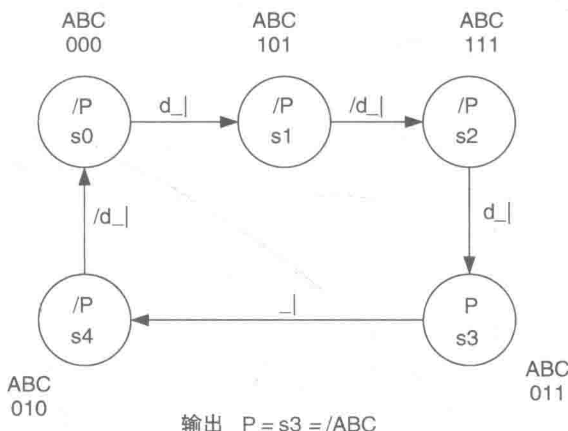


图 2.13　关于输入信号 d 的两次 1 到 0 转变的状态图

在这个状态图中，FSM 观察到信号 d 先被拉高，然后变低（状态 s0 和 s1），然后信号 d 被再次拉高（状态 s2 和 s3）。在状态 s3，状态机得知 d 已经被激活两次，所以输出信号 P 开始输出脉冲。在下一个时钟周期，状态机从 s3 进入 s4，并在 s4 等待信号 d 被拉低，然后回到状态 s0。所以当系统需要输入信号多次被激活的特性时可以用 FSM 来实现。

注意在图 2.13 中，在状态 s0 和 s1，以及 s4 和 s0 之间的转换不是单位距离编码形式，可以试着用单位距离编码来实现状态之间的转换。

一种方法是在 s3 和 s4 之间加一个冗余状态（我们可以称之为 s5），这样就可以实现单位距离编码：s0 = 000，s1 = 100，s2 = 110，s3 = 111，s5 = 011，s4 = 001。

讲稿 2.13 数据序列检测

请大家看图 2.14 所示的例子。

这种检测模块的时序图如图 2.15 所示。

注意到 d 是在每个时钟的上升沿被采样的（如图 2.15 中的箭头所示）。

FSM 也在时钟的上升沿到来时进行状态转变，时序图最下方显示了状态切换的顺序。

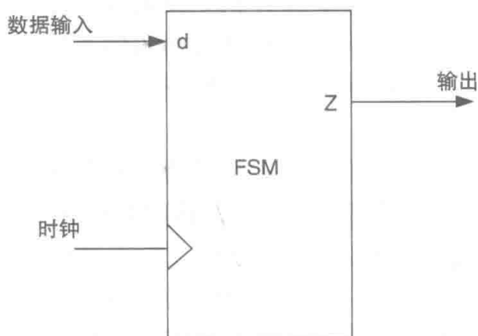


图 2.14 数据序列 101 检测模块框图

图中输入信号 d 的数据序列为 101。当然我们在设计状态机的时候，需要让 FSM 能够识别这个序列，而不是得出其他的结果。只有检测到数据序列 101，输出信号 Z 才产生脉冲。

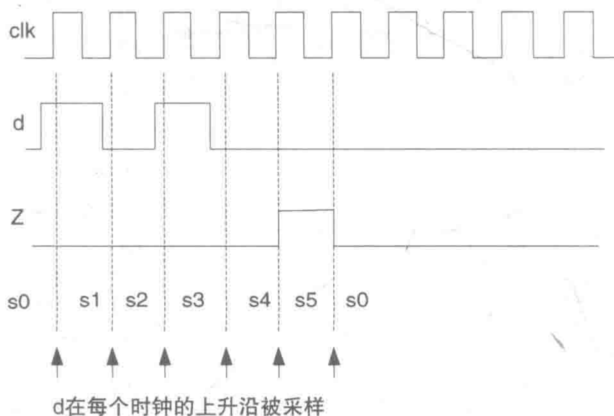


图 2.15 数据序列 101 检测模块时序图

思考题

假设信号 d 输入的是交替序列，即 1→0→1→0，两个脉冲。设计一个可以识别这个序列的 FSM。

提示

首先将图 2.15 识别序列 101 的状态图设计出来，然后根据需要添加必要的状态，使得状态机能够识别 1010，或者其他任何序列。

讲稿 2.14

图 2.16 是检测数据序列 101 的状态图。

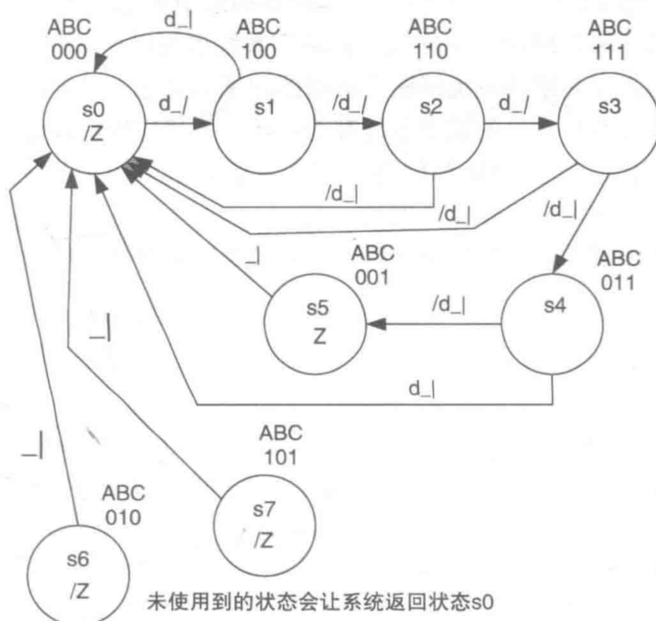


图 2.16　数据序列 101 检测状态图

状态 s0 到状态 s5 负责检测序列 101（注意思考题要求检测 d 端输入的是连续脉冲）。

所有其他返回状态 s0 的路径都表示 FSM 未检测到序列 101 或者出现了其他可能的序列组合。这与图 2.15 中时序图状态变化的情况是一致的。

注意这里检测序列 101 使用了 6 个状态。而二次状态变量允许最多 8 个状态；因此还有 2 个状态没有用到。如果 FSM 无意中进入这 2 个没有定义的状态，系统会出现什么情况？答案就是 FSM 将无法从这 2 个状态中“脱离”，只能停在那里。

为了避免这种情况出现，最常见的做法是在设计 FSM 时将没有用到的状态也考虑到，并定义当 FSM 进入这些状态时不管什么情况都立刻在下一个时钟上升沿到来时回到状态 s0。如图 2.16 所示，FSM 进入状态 s6 和 s7 以后均被引导回到状态 s0。

注意，在使用 D 触发器设计状态机时，当状态机进入未定义的状态时系统会自动复位，状态机将返回状态 s0；因此，这时候没有必要刻意将状态机的未定义状态和状态 s0 相连。后面的章节会有更详细的阐述。

2.3　小结

本章介绍如何使用 FSM 控制外部硬件模块，并组成一个数字系统。后面的章

节将着重介绍使用 FSM 控制各种硬件外部设备的细节问题。如果不具体指明，通常情况下许多基于微处理器（单片机）的数字系统也可以用 FSM 和逻辑电路实现。使用第 1 章和第 2 章介绍的系统框图和状态图的基本概念，结合后面章节所涉及的硬件描述语言，大部分常见的数字系统都能够转化为片内硬件设计。好处在于，和适用微处理器相比，基于 FSM 和硬件描述语言的数字系统消耗更少的硬件资源。读者可以在后续的学习中逐步体会到这一点。

下一步的工作就是关于如何基于状态图生成逻辑电路并实现硬件功能。

第3章　根据状态图综合硬件电路

3.1　关于 FSM 的综合

读到这里，对于 FSM 的设计的各种基本概念都已经介绍完毕了。但是，设计思路需要被付诸实践并能解决一系列的问题，因此下面几章将着重介绍一些应用状态机设计的实例。

在 FSM 设计过程中，我们需要将状态图最终转换成实际的电路，将它们固化到 PLD、FPGA 或者 ASIC（Application Specific Integrated Circuit）芯片中。很明显，这个转换过程有着决定性的意义，但不排除带有一些简单重复的劳动在里面。

FSM 的综合可以在不同的层面来进行：一种是使用触发器来搭建 FSM，触发器的种类可以是 D 触发器、T 触发器和 JK 触发器。还可以使用高级硬件描述语言，例如 VHDL，这相当于将状态图用语言描述的方式直接输入到处理芯片中。运用硬件描述语言结合上述任意一种触发器和一些规则、技巧就能完成系统设计。或者将设计好的状态图转化成 C 语言程序，这样就适合基于微控制器（单片机）系统的解决方案。

无论是运用直接综合的方法还是硬件描述语言，最终设计可以在以下几个平台运行：

- 搭建使用分立 TTL 器件或者 CMOS 器件组成的电路；
- 将程序固化到 PLD 芯片；
- 将程序固化到 FPGA 芯片；
- 将程序固化到 ASIC 芯片；
- 使用更大规模的集成电路芯片。

大部分设计方案都支持 D 触发器和 T 触发器，这些器件在工业自动化设计中得到了广泛的运用。因此本书将着重向大家介绍如何使用 D 触发器和 T 触发器进行工程设计。注意，JK 触发器也是可以用的，但是不在本书的介绍范围内。

本章我们将先向大家介绍如何使用 T 触发器，然后再介绍如何使用 D 触发器设计 FSM。

为什么要选择 T 触发器或者 D 触发器呢？这两个都是如今最常用的触发器。主要原因是 T 触发器可以很方便地通过 D 触发器来实现，而构建 D 触发器只需要 6 个逻辑门（JK 触发器需要 10 个逻辑门）。这就意味着 D 触发器相对于 JK 触发器而言占用更少的芯片资源。另外一个原因是 D 触发器比 JK 触发器更加稳定。

D 触发器的输入端如果接入逻辑 0，那么它将在下一个时钟到来时自动复位。这个自动复位的特点在设计 FSM 时显得十分有用。

3.2 学习资料

讲稿 3.1 T 触发器

一个 T 触发器可以用一个典型的 D 触发器来实现，如图 3.1 所示。

从图中可以看出，T 触发器是由一个 D 触发器加上一个异或门组合而成的，其真值表在示意图的下方。

真值表中，Q_n 是触发器输出的当前状态（在下一个时钟脉冲到来前），而 Q_{n+1} 是触发器输出的下一个状态（时钟脉冲到来之后）。只要输入端 t 为逻辑 1，触发器将在每一个时钟脉冲到来时改变状态，如果 t 为 0，则触发器保持不变。

T输入　时钟　D SET Q　CLR $\overline{Q}$　Q 输出

T	Q_n	Q_{n+1}	
0	0	0	无变化
1	0	1	触发
1	1	0	触发
0	1	1	无变化

图 3.1　T 触发器的示意图和真值表

因此，t 是用来控制触发器的，当触发器需要改变状态时，就将 t 置高，其余时候保持逻辑 0 便可。

讲稿 3.2 T 触发器示例

我们将讲稿 1.13 里的带指示功能的单脉冲发生器的状态图搬到这里，作为图 3.2。

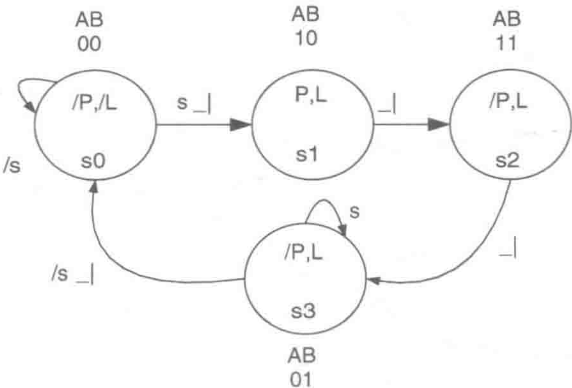


图 3.2　带指示功能的单脉冲发生器状态图

观察二次状态变量 A 的变化，读者可以将 A 的值从 0 到 1 或者从 1 到 0 变化时所处的状态记录下来。如图中所示，从状态 s0 到 s1，A 的值产生了变化，即从 0 变到 1，然后在状态 s2 到 s3 的过程中，A 的值由 1 变到 0。因此我们可以写出如下公式：A · T = s0 · s + s2。

上述公式定义了 T 触发器 A 的输入端 T 的逻辑表达式。

只要 FSM 在状态 s0，当输入信号 s 变为逻辑 1 时，T 触发器 A 的输入端 T 的值将变为逻辑 1。当输入 T 为高，触发器将翻转（切换状态）。而在状态 s0，触发器 A 和 B 都被复位，当 s 被置高，下一个时钟将导致触发器 A 的输出从 0 变为 1（B 保持不变）。

在状态 s1，T 触发器 A 的输入端 T 的值将返回为 0，因为公式里没有关于输入信号 T 在状态 s1 置高的部分。因此，A 的值将不随着时钟脉冲的到来而变化。当 FSM 进入状态 s2 时，T 触发器 A 的输入端 T 将再次被置高，这时，触发器 A 的输出将随着下一个时钟到来时，在状态 s3 输出逻辑 0。注意到当状态机到达 s3 时，触发器 A 的输入端 T 将再次被置 0，因而当 FSM 返回 s0 时，触发器 A 的值保持不变。

注意上述公式里触发器输入端 T 从 0 变为 1 是在 FSM 的当前状态发生的。这一点很必要，它确保了触发器将在下一个时钟到来（即离开当前状态，进入下一个状态）时，输出得到改变。这里改变状态的逻辑电路，指的就是 FSM 的“下一状态解码器”（参考讲稿 1.4）。

思考题

请读者试着将 T 触发器 B 的公式写出来。

讲稿 3.3

T 触发器 B 所对应的输入端 T 的公式：B · T = s1 + s3 · /s。

在状态 s1，输入信号 B · T 需要被置 1，因此在下一个时钟脉冲到来时，触发器将改变其状态，输出从 0 变为 1。注意，在这里状态 s1 和 s2 之间没有外部输入条件进行制约。

当外部输入信号 s 为逻辑 0 且下一个时钟脉冲到来时，公式右边的第二项 s3 · /s 将把 T 触发器 B 的输出在状态 s3 从 1 重置为 0。

实际上，所做的就是寻找触发器输出状态从 0 变化到 1 或者从 1 变化到 0 的情况来进行分析。

思考题

现在根据图 3.3，将每个输出信号的公式写出来。输出信号 L 的值在 s3 变高的条件是外部输入信号 R 的值为逻辑 1。

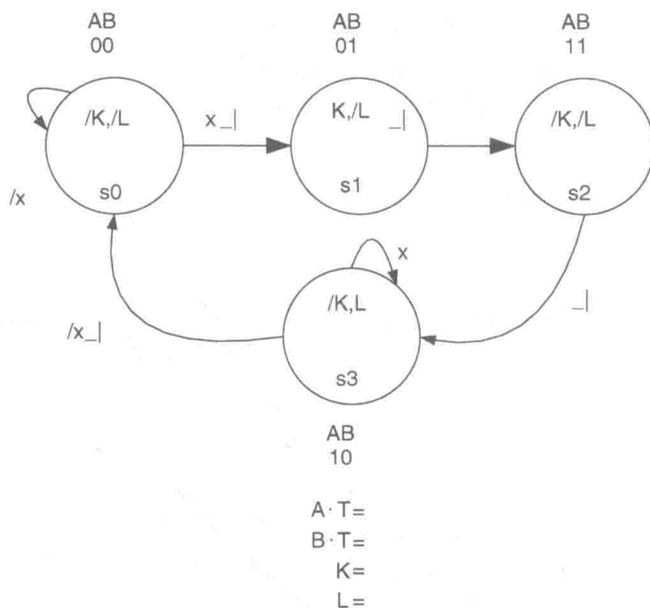


图 3.3 使用 T 触发器构建的状态图

讲稿 3.4

修改后的状态图如图 3.4 所示。

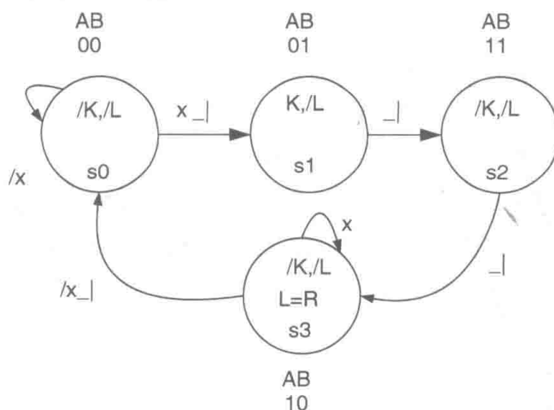


图 3.4 使用 T 触发器构建的状态图

触发器 A 和 B 所对应的公式分别是 A · T = s1 + s3 · /x，B · T = s0 · x + s2。

外部输出信号的公式为 K = s1 = /A · B，L = s3 · R = A · /B · R。

L 属于米利（Mealy）型输出，它输出逻辑 1 必须满足两个条件：一个是状态机处于状态 s3，另一个是输入信号 R 的值必须是逻辑 1。如果对公式 L 的推导方法不是很明确，读者可以复习讲稿 3.1 ~ 3.3。

讲稿 3.5

思考题

完成下面几个题目。写出每个状态图所对应的触发器公式和输出信号公式。如果觉得有困难，先学习讲稿 3.1 ~ 3.4。

讲稿 1.19 中图 1.22 所对应的状态图，其中使用的二次状态变量如下：

	ABC
s0	000
s1	100
s2	110
s3	011
s4	001

讲稿 2.3 中图 2.4 所对应的状态图，其中使用的二次状态变量如下：

	AB
s0	00
s1	10
s2	11
s3	01

讲稿 2.12 中图 2.13 所对应的状态图，其中使用的二次状态变量如下：

	ABC
s0	000
s1	100
s2	110
s3	111
s4	011
s5	001

讲稿 2.9 中图 2.10 所对应的状态图，其中使用的二次状态变量如下：

	ABCD
s0	0000
s1	1000
s2	1100
s3	1110
s4	1111
s5	0111
s6	0011
s7	1011
s8	1001
s9	0001

讲稿 3.6

讲稿 3.5 中思考题的答案如下所示：

讲稿 1.19 中图 1.22 所对应的状态图：

	ABC	答案
s0	000	A · T = s0 · s + s2 = /A/B/C · s + AB/C
s1	100	B · T = s1 + s3 = A/B/C + /ABC
s2	110	C · T = s2 + s4 · /s = AB/C + /A/BC · /s
s3	011	
s4	001	P = s1 + s3 · x = A/B/C + /ABC · x

讲稿 2.3 中图 2.4 所对应的状态图：

	AB	答案
s0	00	A · T = s0 · st + s2 · /to = /A/B · st + AB · /to
s1	10	B · T = s1 + s3 · /st = A/B + /AB · /st
s2	11	
s3	01	P = s1 + s2 = A，TS（低有效）= /s1 = /（A/B）

讲稿 2.12 中图 2.13 所对应的状态图：

	ABC	答案
s0	000	A · T = s0 · d + s3 = /A/B/C · d + ABC
s1	100	B · T = s1 · /d + s4 · /d = A/B/C · /d + /ABC · /d
s2	110	C · T = s2 · d + s4 · /d = AB/C · d + /ABC · /d
s3	111	
s4	011	P = s3 = ABC

讲稿 2.9 中图 2.10 所对应的状态图：

	ABCD	答案
s0	0000	A · T = s0 · int + s4 + s6 + s8 · f
s1	1000	= /A/B/C/D · int + ABCD + /A/BCD + A/B/CD · f
s2	1100	B · T = s1 + s5 = A/B/C/D + /ABCD
s3	1110	C · T = s2 · eoc + s7 = AB/C/D · eoc + A/BCD
s4	1111	D · T = s3 · /eoc + s8 · /f + s9 · /int
s5	0111	= ABC/D · /eoc + A/B/CD · /f + /A/B/CD · /int
s6	0011	

（续）

	ABCD	答案
s7	1011	
s8	1001	RC = /s0 = / (/A/B/C/D) 低有效输出
s9	0001	S/H = s1 + s2 = A/C/D
		SC = s2 = AB/C/D
		CS = / (s3 + s4 + s5) = / (ABC + BCD) 低有效输出
		W = /s4 = / (ABCD) 低有效输出
		CC = s7 = A/BCD

如果你觉得理解低有效输出信号有一些困难，可以翻到后面先学习讲稿 3.25 和讲稿 3.26，然后再回到这里。

思考题

现在请大家试着将讲稿 2.14 中图 2.16 所对应的各个公式写出来。二次状态变量已经在图中标出。答案在讲稿 3.7 中。

讲稿 3.7

讲稿 3.6 中的思考题所对应的各个公式如下：

A · T = s0 · d + s1 · d + s2 · /d + s3 + s7

= /A/B/C · d + A/B/C · d + AB/C · /d + ABC + A/BC

= /B/C · d + AB · /d + AC

B · T = s1 · /d + s2 · /d + s3 · d + s4 + s6

= A/C · /d + BC · d + /AB

C · T = s2 · d + s3 · d + s4 · d + s5 + s7

= AB · d + BC · d + /BC

Z = s5 = /A/BC

如何用 T 触发器，构建并综合一个 FSM 的全过程，到此全部介绍完毕。

在讲稿 3.1 中，曾经提到 T 触发器是由 D 触发器加上一个异或门组成的。一些 PLD 芯片对这两种触发器都支持，因此 FSM 的设计可以通过使用这些 PLD 芯片来完成。

另外，一些 PLD 器件可以通过编程被用作这两种触发器的其中一种。然而，大部分 PLD 器件只支持 D 触发器，特别是那些价格低廉的器件，例如 22v10。因此，我们有必要讨论如何用 D 触发器来设计并综合 FSM。

事实上，使用 D 触发器设计 FSM 的技巧性更强一些，需要注意的地方也比较多，因此花费的篇幅也比较多。当然花时间学习使用 D 触发器设计 FSM 肯定是值得的，因为这使得相当一部分只支持 D 触发器的器件，成为潜在的 FSM 设计使用

对象。

讲稿 3.8 使用 D 触发器综合 FSM：D 触发器公式

图 3.5 给出了基本的 D 触发器的示意图。

D 触发器只有一路输入信号 D（当然时钟也是输入信号）。

- 输入信号必须在时钟脉冲到来之前被置高，这样输出端 Q 才能在时钟脉冲到来时输出逻辑 1。
- 如果输出 Q 需要保持逻辑 1，输入端 D 必须一直置高，这样在下一个时钟脉冲到来时，触发器会继续将 D 的值输出到 Q 端。

这两个要点在使用 D 触发器时至关重要，大家必须牢记。

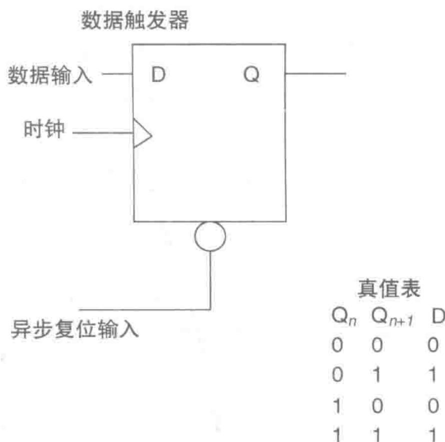


图 3.5 D 触发器示意图和特性

图 3.6 给出了 D 触发器的时序图，请注意这张图并不完整。

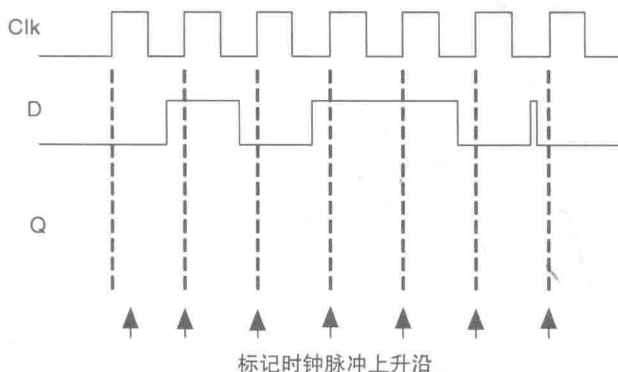


图 3.6 D 触发器时序图（不完整版）

思考题

将时序图中输出信号 Q 的部分补充完整。

提示

可以反复学习并理解本节的内容后，再试着补齐这张图。

讲稿 3.9

图 3.7 给出了上一节留出的思考题的答案，也就是 D 触发器时序图的完整版。

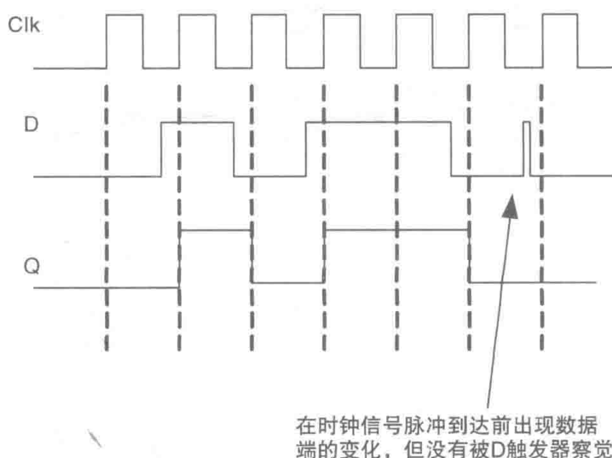


图 3.7　完整的 D 触发器时序图

重点是当时钟信号从 0 变为 1（即脉冲到来）时，留意输入端 D 的值；此时不管 D 的值是什么，输出端 Q 的值和此刻 D 的值相同。

因为 D 触发器的特性就是在时钟脉冲到来那个时刻在输出端 Q 输出 D 的值。

图中可以看到，输入信号 D 被置高长达两个时钟周期。这意味着对应 Q 的值也会被置高两个时钟周期。

同时也要注意到输入端 D 的值在两个时钟脉冲之间被置高很短暂的时间，对于这种情况，触发器无法辨别这么短的脉冲，因此在输出端就没有任何反应。

在此必须要重复一下：触发器只有在时钟脉冲到来时才会更新输出端的值。

讲稿 3.10

在学习了 D 触发器的基本特性之后，让我们再来看图 3.8 所对应的状态图。

当然这张图在讲稿 1.13 里也出现过，就是那个带指示功能的单脉冲发生器。但这次使用 D 触发器来设计状态机。

触发器 A 所对应的公式为 A · D = s0 · s + s1 = /A · /B · s + A · /B = /B · s + A/B（辅助定律）。

触发器 A 的输入端 D 必须在状态 s0 时被置 1，而且在状态 s1 里必须保持逻辑 1。

触发器 B 所对应的公式为 B · D = s1 + s2 + s3 · s = A · /B + A · B + /A · B · s = A + /A · B · s = A + B · s。

公式的第一项将触发器 B 的输入端 D 置高，对应状态 s1。而公式的第二项是在状态 s2 让输入端 D 的值保持逻辑 1。但是公式的第三项意味着什么？

在状态 s3，如果输入信号 s 的值为逻辑 1，则触发器 B 的输入端 D 需要保持逻

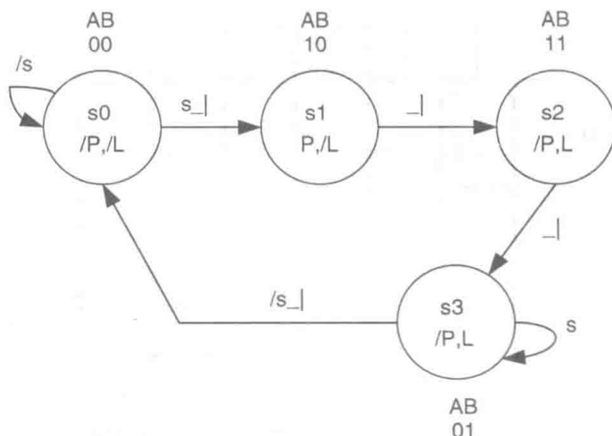


图3.8 使用D触发器构建的状态图

辑1，因为当s=0时，FSM的任务是回到初始态（复位触发器B），即状态s0。因此，只有当s保持逻辑1时，公式的第三项才是有效的，而当s=0时，这一项为0，触发器B被复位，FSM回到s0。

当FSM到达状态s3时，下一个状态将是回到s0，因此对于状态s3的输入信号应该是/s，但这里将输入信号取反（变为s）并和状态s3一起，让D触发器的输入端保持逻辑1。

法则1 状态转换过程中出现触发器输出从1变为0的情况，需要将此时的状态转换对应的输入信号取反并和当前状态相与（成为与门的两个输入端）。

讲稿3.11

现在请大家看图3.9所示的状态图。

它将之前的单脉冲发生器做了一些改动，如果输入信号k=1，m=1，FSM则产生多个脉冲，如果k=1，m=0，FSM会每隔4个时钟周期产生多个脉冲。

触发器A所对应的公式为A·D=s0·s+s1+s2·m。

公式的第一项代表在一开始将触发器的输入置1，以便在下一个时钟脉冲到来时触发器的输出为逻辑1，且FSM进入状态s1。

公式的第二项是让触发器在状态s1和s2之间保持逻辑1的输出。但必须注意到s1和s2之间的输入信号k并没有出现在公式里，原因在于没有必要。不管k的值是0还是1，触发器总是输出逻辑1。

法则1 当触发器的输出在前后两个状态转换时始终为逻辑1，公式里对应的项不需要带有状态转换时的输入信号。

公式的第三项稍微复杂一点。它是状态s2的保持项。在状态s2时，从s2到s3，触发器A的输出从1变为0。因此，这里用法则1（在讲稿3.10中已经定义）来解释。另一种情况是从s2回到s1，触发器输出为1到1，没有变化。不过，从

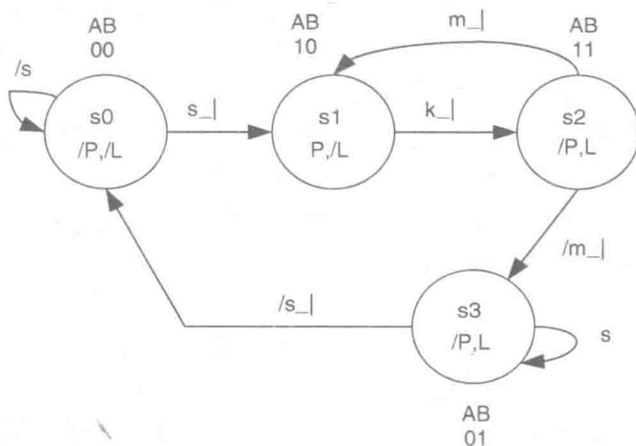


图 3.9　带两路分支的状态图

s2 到 s1 的情况在公式里并没有体现。因为没有必要，所以得出如下的法则 3。

法则 2　一个带两路分支的状态转换，当触发器的输出变化分别为 1 到 0 和 1 到 1 时，通常只保留 1 到 0 的转换作为公式项，同时将转换时的输入信号取反并和状态本身相与，且不用考虑触发器输出 1 到 1 的情况。

讲稿 3.12

为了更好地理解这 3 个法则，将讲稿 3.11 中的图 3.9 重新画出来，变为图 3.10，方便大家进一步理解。

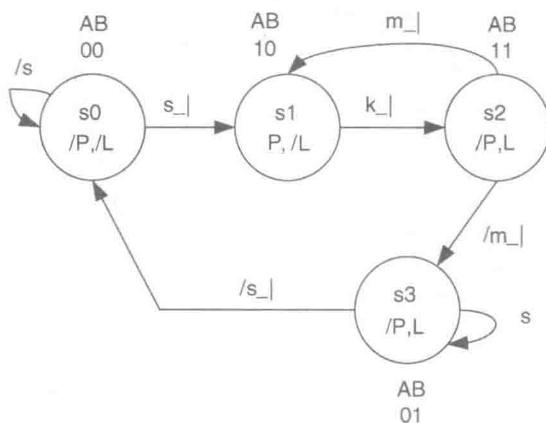


图 3.10　带两路分支的状态图

法则 1　状态转换过程中出现触发器输出从 1 变为 0 的情况，需要将此时的状态转换对应的输入信号取反并和当前状态相与（成为与门的两个输入端）。

当两个状态之间的转换带有触发器输出从 1 到 0 的情况，且有输入信号作为条件时，需要将输入信号取反并和对应的状态相与，目的是在输入信号的值满足触发状态转换之前将触发器的输出保持为逻辑 1。

当 1 到 0 的转换过程中没有附带任何输入条件时，这样的情况没有必要出现在公式里，因为 FSM 会自然地往下走，触发器也会随之被复位。

法则 2 当触发器的输出在前后两个状态转换时始终为逻辑 1，公式里对应的项不需要带有状态转换时的输入信号。

上述状态图中，状态 s1 和 s2 之间的转换对于触发器 A 来说，表达式可以写成 s1 · k + s1 · /k，即不管 k 的值是什么，在 s1 和 s2 状态中，触发器 A 的输出均为逻辑 1。因此运用布尔逻辑运算法则将 s1 · k + s1 · /k = s1 简化是成立的。

法则 3 一个带两路分支的状态转换，当触发器的输出变化分别为 1 到 0 和 1 到 1 时，通常只保留 1 到 0 的转换作为公式项，同时将转换时的输入信号取反，再和状态本身相与，且不用考虑触发器输出 1 到 1 的情况。

上述状态图中触发器 A 在状态 s2 带有两路分支，一个是 1 到 0 的转换（s2 到 s3），另一个是 1 到 1 的转换（s2 到 s1）。在状态 s2，如果输入信号 m = 1，触发器 A 将保持逻辑 1 的输出，否则当 m = 0 时，触发器必须复位。

讲稿 3.13

图 3.11 列出了状态图中两路分支所有可能涵盖的条件。

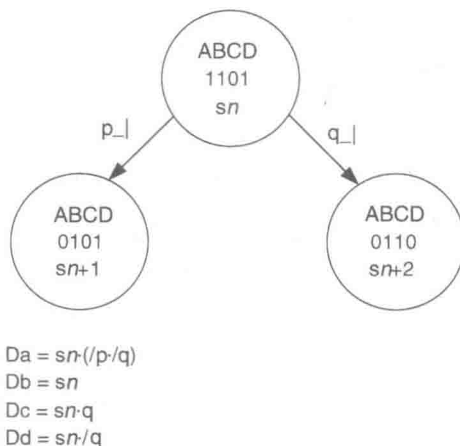


图 3.11 两路分支状态图所包含的所有条件

特别要注意的是，图中触发器 A 的输出均为 1 到 0。公式 DA = *sn* · (/p · /q) 中就能很明显地体现输入信号是被取反后加入的。只有当输入信号 p = 0 及 q = 0 这两个条件都满足时，FSM 才会停留在状态 *sn*。

讲稿 3. 14

请大家看图 3. 12 所示的状态图。

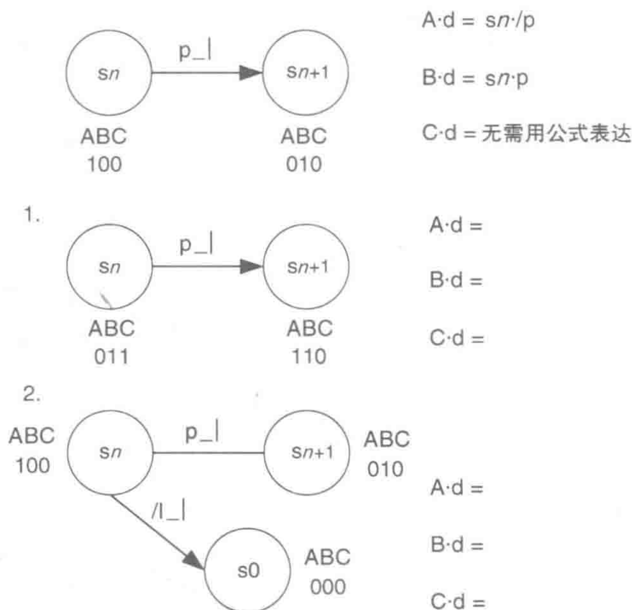


图 3. 12　带两路分支的状态图示例

思考题

完成图中两组 D 触发器的公式。

讲稿 3. 15

讲稿 3. 14 中思考题的答案如下：

1. A · D = s*n* · p

   B · D = s*n*

   C · D = s*n* · /p

2. A · D = s*n* · /p · l，状态机停留在 s*n* 的条件是 p = 0 和 l = 1 都要满足；

   B · D = s*n* · p，状态 s*n* 和 s*n* + 1 之间是一个 0 到 1 的转换；

   C · D，没有触发器 C 所对应的公式项。

请大家复习一下讲稿 3. 8 ~ 3. 14 所涵盖的内容，特别是状态机公式的推导方法，然后完成下面的思考题。

思考题

状态图如图 3. 13 所示，要求用 D 触发器来完成设计和综合，其中有两个状态

是带有两路分支的。写出触发器 A 和 B 所对应的公式，并写出输出信号 X 的公式。

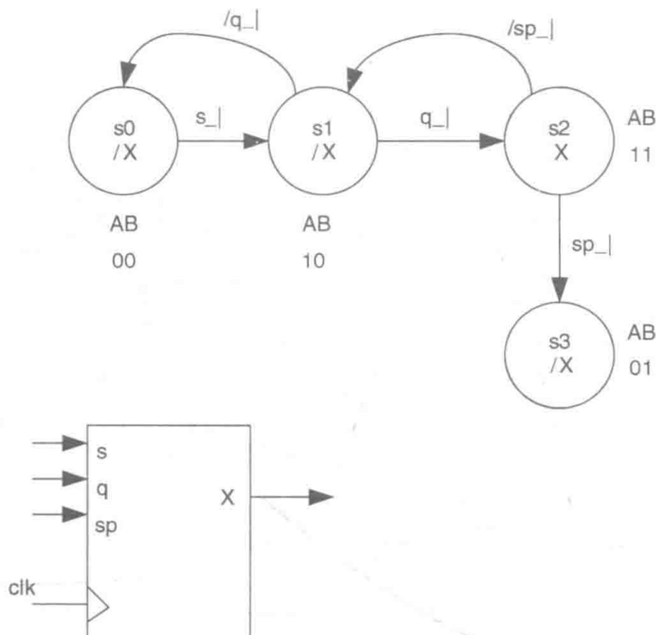


图 3.13 多个两路分支状态示例

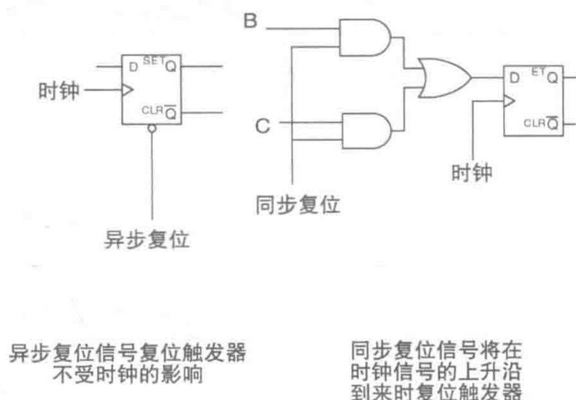
讲稿 3.16

讲稿 3.15 中思考题的答案为 A · D = s0 · s + s1 · q + s2 · /sp，对于触发器 A，s0 · s 是一个置高的项，s1 · q 是从 s1 到 s0 的一个 1 到 0 的转换，s2 · /sp 是从 s2 到 s3 的一个 1 到 0 的转换。B · D = s1 · q + s2 · sp + s3，对于触发器 B，s1 · q 是一个置高的项，s2 · sp 是从 s2 到 s1 的一个 1 到 0 的转换，s3 是状态保持项。大家必须注意，FSM 到了 s3 之后会锁死。FSM 输出信号 X 的表达式为 X = s2，而且类型为摩尔型，因为它是一个关于二次状态变量的函数。如果要离开状态 s3，系统必须带有初始化功能，有必要植入一个复位输入信号。因此在任何情况下，FSM 都必须带有初始化功能。

触发器的复位

如果触发器带有异步复位输入信号（见图 3.14），只要将所有的异步复位信号连在一起便能够复位所有的触发器。

如果触发器没有异步复位输入信号（或其他任何复位信号），可以通过将触发器的输入端 D 和一个外部的复位信号组成与门完成复位功能。在输入信号为同步的情况下，复位信号（低有效）正常情况下都是被拉高的；这样同时也让所有触发器的数据输入信号变得有效。如果将复位信号拉低，D 触发器的输入端也会被拉



$D = (B + C) \cdot reset$

图 3.14　D 触发器的同步和异步复位

低，在下一个时钟到来时，触发器会被复位。提醒一下，如果触发器是靠上升沿触发的，则其复位将在时钟的上升沿到来时完成。

讲稿 3.17

思考题

试着写出每一个状态真值表所对应的 D 触发器公式。如果仍然有许多地方不确定，可以先复习讲稿 3.8 ~ 3.16 的内容。

讲稿 1.19 中的状态表，对应图 1.22，二次状态变量和状态对应关系如下：

	ABC
s0	000
s1	100
s2	110
s3	011
s4	001

讲稿 2.3 中的状态表，对应图 2.4，二次状态变量和状态对应关系如下：

	AB
s0	00
s1	10
s2	11
s3	01

讲稿 2.12 中的状态表，对应图 2.13，二次状态变量和状态对应关系如下：

	ABC
s0	000
s1	100
s2	110
s3	111
s4	011

讲稿 2.9 中的状态表，对应图 2.10，二次状态变量和状态对应关系如下：

	ABCD
s0	0000
s1	1000
s2	1100
s3	1110
s4	1111
s5	0111
s6	0011
s7	1011
s8	1001
s9	0001

讲稿 3.18

讲稿 3.17 中思考题各个图表所对应的答案如下：

讲稿 1.19，图 1.22 状态表所对应的方程式：

	ABC	答案
s0	000	AD = s0 · s + s1 = /A/B/C · s + A/B/C = /B/C · s + A/B/C
s1	100	BD = s1 + s2 = A/C
s2	110	CD = s2 + s3 + s4 · s = AB/C + /ABC + /A/BC · s = AB/C + /ABC + /AC · s
s3	011	
s4	001	P = s1 + s3 · x = A/B/C + /ABC · x　其中 x 是输入信号

讲稿 2.3，图 2.4 状态表所对应的方程式：

	AB	答案
s0	00	AD = s0 · st + s1 + s2 · to = /B · st + A/B + A · to
s1	10	BD = s1 + s2 + s3 · st = A + B · st
s2	11	
s3	01	P = s1 + s2 = A
		TS = /s1 = /(A/B) 低有效输出

讲稿 2.12，图 2.13 状态表所对应的方程式：

	ABC	答案
s0	000	AD = s0 · d + s1 + s2
s1	100	= /A/B/C · d + A/B/C + AB/C
s2	110	= /B/C · d + A/B/C + AB/C
s3	111	= /B/C · d + A/C
s4	011	
		BD = s1 · /d + s2 + s3 + s4 · d
		= A/B/C · /d + AB/C + ABC + /ABC · d
		= A/C/d + AB + BC · d
		CD = s2 · d + s3 + s4 · d
		= AB/C · d + ABC + /ABC · d
		= AB · d + ABC + BC · d
		P = s3 = ABC

讲稿 2.9，图 2.10 状态表所对应的方程式：

	ABCD	答案
s0	0000	AD = s0 · int + s1 + s2 + s3 + s6 + s7 + s8 · f
s1	1000	= /B/C/D · int + A/C/D + AB/D + /BCD + A/BD · /f
s2	1100	BD = s1 + s2 + s3 + s4
s3	1110	= A/B/C/D + A/C/D · E + ABC
s4	1111	= A/C/D + ABC · s
s5	0111	CD = s2 · eoc + s3 + s4 + s5 + s6
s6	0011	= B/D · eoc + ABC + BCD + /ACD
s7	1011	DD = s3 · /eoc + s4 + s5 + s6 + s7 + s8 · f + s9 · int
s8	1001	= ABC · /eoc + CD + A/BD · f + /A/BD · int
s9	0001	
		RC = /s0 = /(/A/B/C/D)
		S/H = s1 + s2 = A/C/D
		SC = s2 = AB/C/D
		CS = /(s3 + s4 + s5) = /(ABC + BCD)
		W = /s4 = /(ABCD)
		CC = s7 = A/BCD

注：输出信号里低有效的信号显示方式是在相应的状态前面加上非门的斜杠标记。

思考题

当上述题目全部完成后，再将讲稿 3.10，对应图 3.8 的单脉冲发生器中 D 触发器的公式和输出公式完成，最后用 D 触发器组建一个 FSM 的电路图，要求带有异步复位输入信号。

讲稿 3.19

完整的带指示功能的单脉冲发生器设计方案如下：

触发器和输出信号公式：A · D = s0 · s + s1 = A/B + /B · s；B · D = s1 + s2 + s3 · s = A + B · s；P = s1 = A/B；L = B。

对应的电路图在图 3.15 中给出，指示功能（触发器）、输入解码单元（A · D 和 B · D）以及输出解码单元（输出信号 P）等。

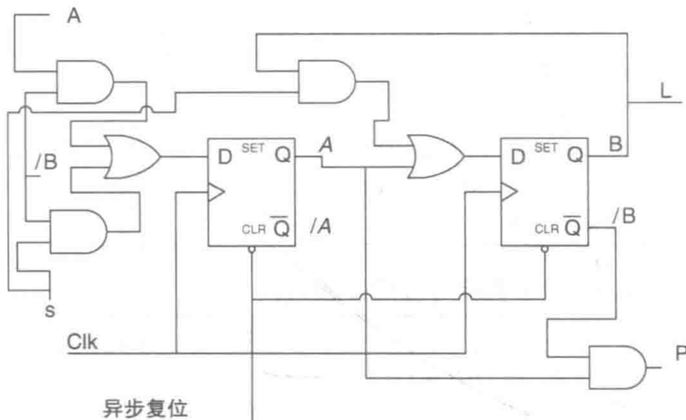


图 3.15 带指示功能的单脉冲发生器（含异步复位）电路图

如果异步复位功能不能实现，那么可以将 A 和 B 的输入端的与门分别添加一个输入信号作为复位，也就是形成同步复位的功能，如图 3.16 所示。

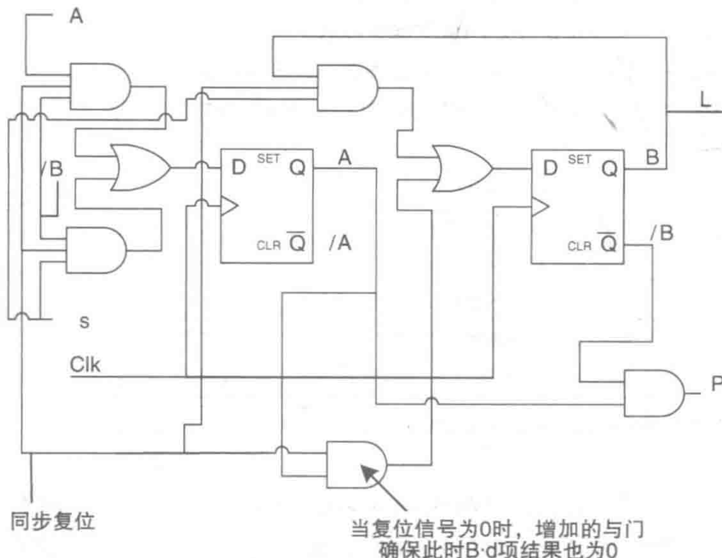


图 3.16 带指示功能的单脉冲发生器（含同步复位）电路图

图 3.16 中，复位信号是每个与门的其中一个输入端。当复位信号有效时，即 reset = 0，必须在触发器 B 的输入端再接入一个与门，确保此时触发器 B 的输入端的信号也为 0，即 B · D = 0。

讲稿 3.20

关于设计 FSM 的各种方法到这里就全部介绍完毕了。从初始化，到设计状态图，再到综合电路图来完成 FSM 的设计等。现在有必要将前面介绍的所有内容在一个工程里向大家完整地演示一遍，以便读者对 FSM 的设计有一个较为全面的认识。现在还是请大家看之前介绍过的单脉冲发生器。

功能描述

首先构建的是一个带有输入和输出的系统框图（见图 3.17）。系统框图一般会作为对系统功能描述的补充。

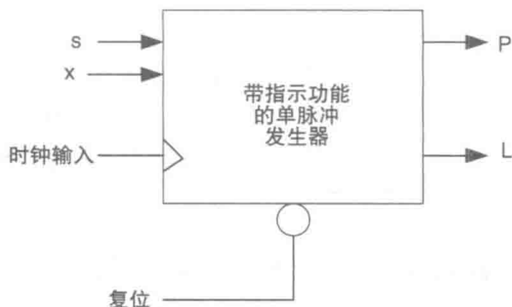


图 3.17 带指示功能的单脉冲发生器系统框图

这里，FSM 的任务是当输入信号 s 为高时 P 端输出一个单脉冲。在信号 s 出现先被拉低，再被拉高这种变化时，系统不应该输出脉冲信号。除此之外，输出信号 L 的作用是作为 P 端的指示位，当它为高时代表 P 端有信号输出。当 s 为低时，P 会输出逻辑 0。输出 L 可以通过输入信号 x 置零（x = 0）来拉低。

下一步是建立状态框图。这一步至关重要，因为它需要一些技巧，并且在整个工程的设计中占有很重要的比例。

讲稿 3.21

系统状态框图如图 3.18 所示。

这里需要给状态图标记二次状态变量，接着写出每个触发器和输出信号的公式。

触发器和输出信号公式

$A \cdot D = (s0 \cdot s + s1) \cdot reset = (A/B + /B \cdot s) \cdot reset$

$B \cdot D = (s1 + s2 + s3 \cdot s) \cdot reset = (A + B \cdot s) \cdot reset$

$P = s1 = A/B$

$L = s2 \cdot x + s3 \cdot x = B \cdot x$

最终，根据公式，可以画出电路图（见图 3.19）。注意输出 L 是米利（Mealy）型输出，因为它的状态受到输入信号 x 的影响。

随后可以运行仿真软件来验证原有的设计方案是否正确。

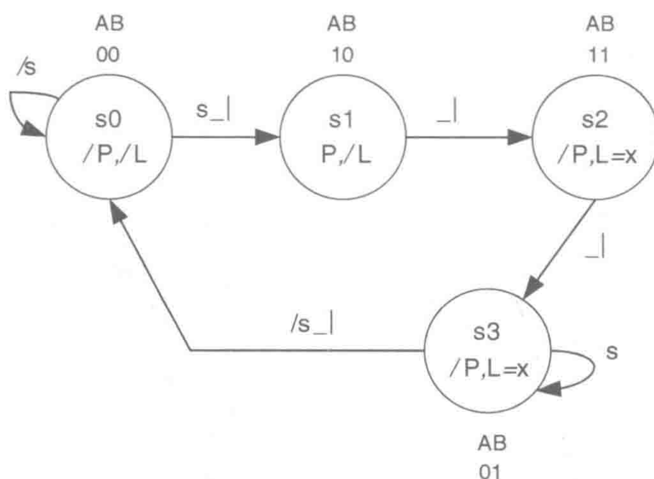


图 3.18 带指示功能的单脉冲发生器状态图

仿真

注意到输出 L 受到输入信号 x 的制约，因此只有在状态 s2 和 s3 时才为逻辑 1，并且此时输入信号 x 必须为逻辑 1。仿真波形如图 3.20 所示。

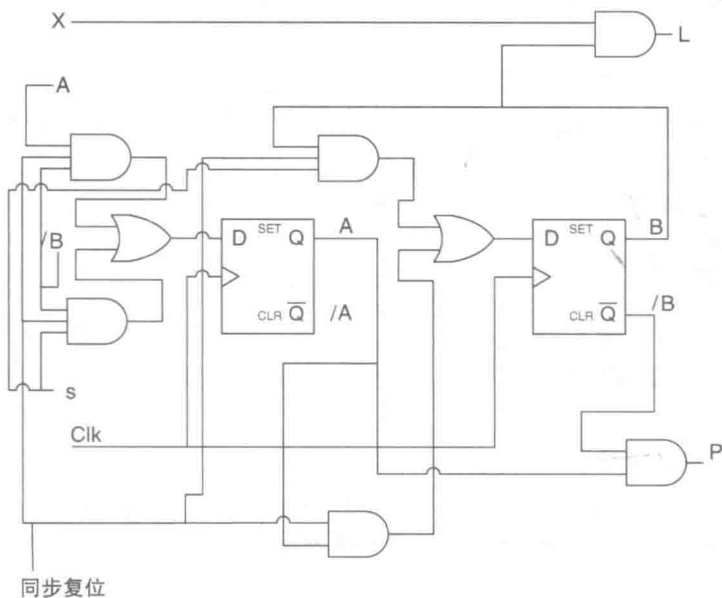


图 3.19 带指示功能的单脉冲发生器电路图

讲稿 3.22

某些情况下，状态机可能会出现三路（或者多路）分支。这种情况之前还未涉及，但是原理是共通的，每一路都可以用前面介绍的设计法则来完成。然而，条件是每一路和其他分支都是相对独立的。

请大家看图 3.21。

触发器 A 的输入项 A・d 在三路分支都是 0 到 1 的状态转换。根据 D 触发器的特性，传输线上的变化条件（x、y 和 z）必须参与一个逻辑或计算，确保在其中任何一个条件满足的情况下，触发器都能顺利输出。

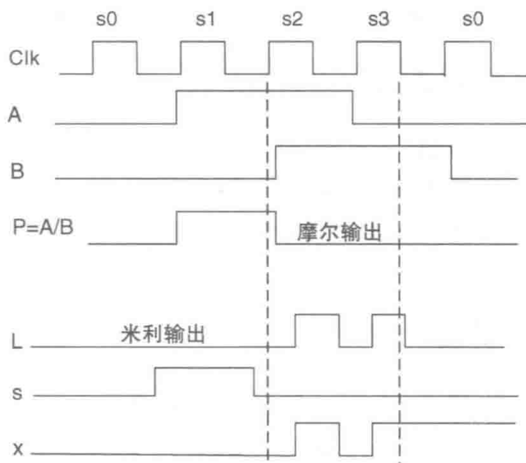


图 3.20　带指示功能的单脉冲发生器仿真时序

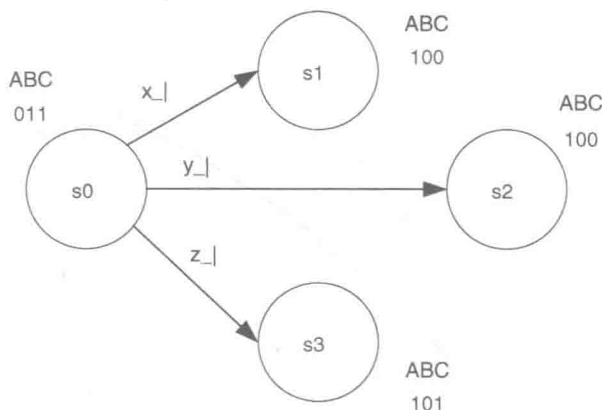


图 3.21　三路分支状态图

对于触发器 B，三路均为 1 到 0 的状态转换。这种情况的处理方法是运用前面介绍的 1 到 0 转换所需要的法则 1。

触发器 C 的情况是两个 1 到 0 转换和一个 1 到 1 的转换。这里处理的方法是，对于两个 1 到 0 的转换，法则 1 仍然是有效的方法。两个输入信号取反后，和当前状态相与，这样 FSM 会在与门输出为逻辑 1 的情况下保持在状态 s0。至于 1 到 1

的转换，通常情况下，是忽略的。

讲稿3.23

思考题

请读者思考图3.22给出的状态机分支图，写出三个触发器的公式A·d、B·d和C·d。

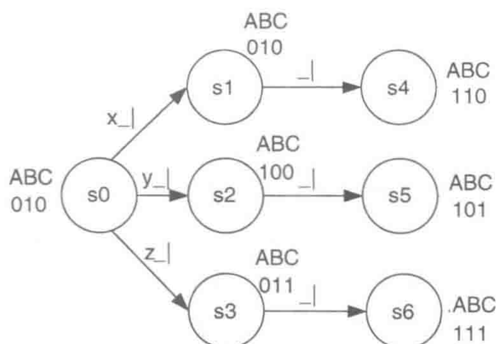


图3.22 三路分支状态图思考题

讲稿3.24

讲稿3.23中三路分支的答案如图3.23所示。

触发器B的公式中，s0·/y项作用是将FSM保持在状态s0。触发器C的公式中，s0·z项的作用是在z的值变为逻辑1之前将FSM保持在状态s0。这些均可以在之前提到的3个法则里找到答案。

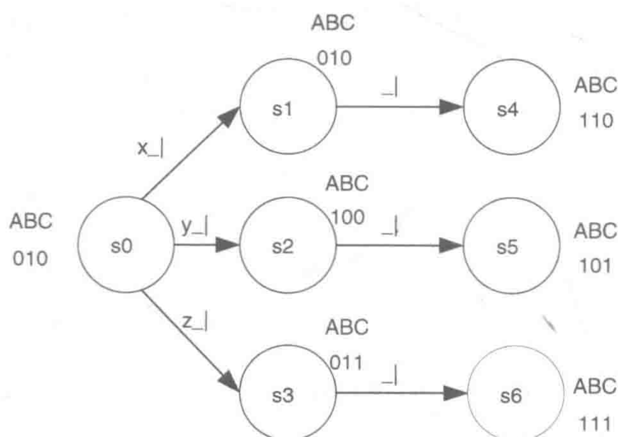


图3.23 三路分支状态图思考题对应答案

讲稿3.25 如何处理多路摩尔（Moore）型低有效输出信号

在设计某些状态机的过程中，需要将输出公式以“低有效”的形式来表达，

而不是“高有效”。这种情况在控制一些存储器件时会经常碰到，例如许多存储芯片的片选（CS）信号通常都是低有效的。如果这个信号被设计为高有效，那么整个 FSM 里所有牵涉到这个信号的状态为“释放”的情况都必须被写进表达片选信号的公式里，这会导致设计的烦琐和难以理解（公式也会很复杂）。

图 3.24 给出了一个典型的例子。

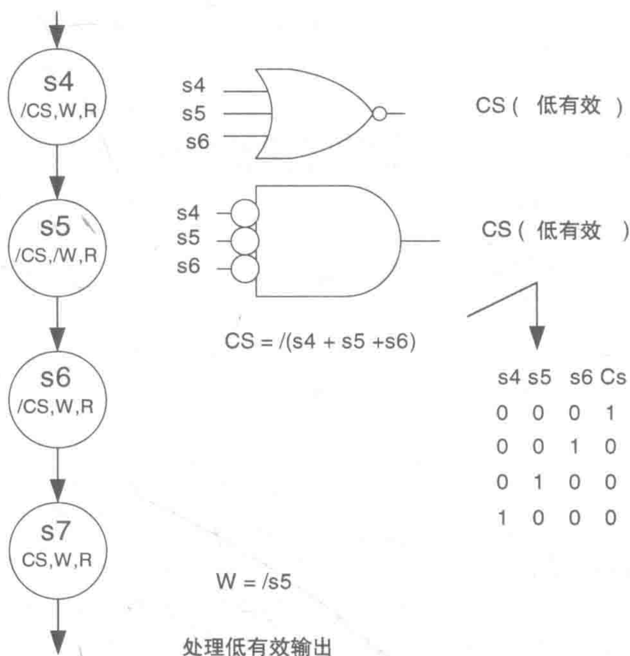


图 3.24　处理低有效的输出信号

图中，CS 信号在状态 s4、s5 和 s6 时均有效（值为逻辑 0），但到了状态 s7 回到了高位。

3 个状态的处理方式主要有两种，一种方式是将 3 个状态作为一个或门的输入，在或门的输出端取反（或非门），代表低有效；另一种方式是用 De Morgan 法则将 3 个状态分别取反后输入一个与门。

讲稿 3.26

现在有这么一种情况，某个状态机的其中一个低有效输出信号，它有一些限定条件，例如只有在其中一个状态下，并且在某个特定输入信号为低（或者高）时才有效（米利型低有效输出）。图 3.25 给出了上述假设的状态图。

在状态 s5，输出 W 的表达式为/W = /x，这意味着，在状态 s5，W 的值为逻辑 0，但这种情况只有在状态 s5 才出现，并且附带条件是 x 的值为逻辑 0。

当我们写 W 的公式时，还需要将状态 s5 纳入表达式：W = /(s5 · /x)，注意

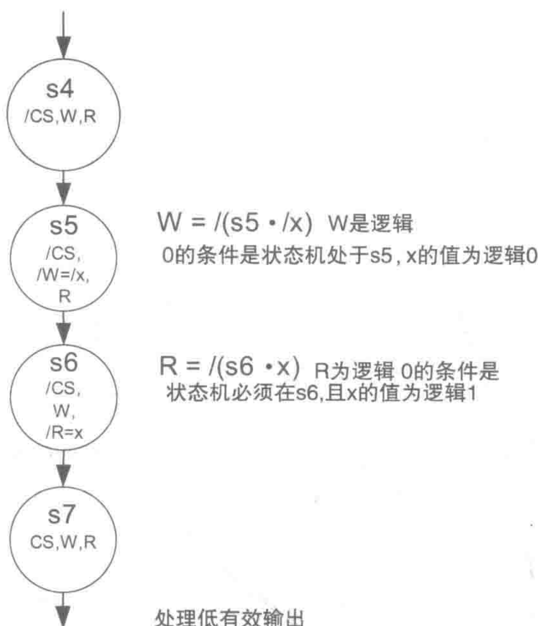


图 3. 25 处理低有效输出信号

到公式的右半边被整个取反用来表示 W 为低有效信号。

用同样的方法，在状态 s6，输出信号 R 可以表示为/R = x，意味着在状态 s6，只有当输入信号 x 为逻辑 1 时，输出信号 R 的值变为逻辑 0。公式可以写为 R = /(s6 · x)，公式右边也同样运用了整体取反的形式来表示低有效。

3.3 小结

本章主要介绍如何运用状态图来综合工程并得出电路图。其中 T 触发器和 D 触发器的合理运用将使得整个设计流程变得简便和高效。在第 4 章的一些例子中，它们也将继续得到运用。

同步 FSM 的设计方法至此已经全部介绍完毕，后续章节将重点关注一些传统的设计模式。

还有一种设计同步 FSM 的方法在本章没有涉及，即“独热编码”技术。它将在第 5 章中单独介绍。

第4章　同步FSM设计

本章主要向大家介绍一些工程案例，并运用前3章所学的知识进行分析和综合。对比传统的FSM设计方法和本书介绍的一些技巧，读者可以清晰地认识到后者在实际运用过程中更加高效。传统方法已经在众多关于数字电路设计的教科书中频繁出现，它主要运用状态转换表格进行FSM的设计，但是当系统带有多个输入信号时，这种方法就显得很繁杂。即使不出现多个输入信号的情况，本书介绍的方法仍然比传统方法快捷和易用。

大部分设计师习惯运用冗余的二次状态变量来减少系统的触发器使用数量和优化输出公式。本章中也会对这种方法给予阐释，并给出一些有趣的结果供大家思考。

本章同时还介绍了一些系统级的设计方案。其中有一部分带有最终的仿真结果。Verilog HDL的源代码在本章不会向大家提供，如何编写Verilog HDL代码将会在本书的后半部分详细介绍。

本章一共包含8个案例，向读者展现了如何运用不同的方法设计带有各种要求的系统。

4.1　传统状态图的综合方法

在运用前3章的内容设计系统之前，有必要介绍一下传统的FSM的设计方法。然后将两种方法进行对比，按照常理，它们的最终结果应该是一样的，或者说系统的复杂度（即逻辑门的个数）应该是相似的。

请大家参考图4.1所示的状态图。系统含有4个状态，需要两个D触发器。使用常规的方法，需要从设计状态表开始，其中包括输入信号x所有可能的情况，触发器A和B的当前状态（PS）和下一个状态（NS）等。表格还附加了两个触发器输入端Da和Db根据输入信号x的变化所形成的下一个状态变化情况。这部分描述在表4.1里得到了体现。

表格中A和B的值是根据图4.1中触发器的状态获得的。例如，在状态s0（AB的当前状态为AB=00）的情况下，当x=0时，AB在下一个状态的值为00；如果x=1，则AB的下一个状态为01（即进入状态s1）。

而Da和Db对应的下一个状态的值和AB所对应的值是相同的，原因是D触发器的输出总是和输入相同的。

请大家将表格的其余部分按照上述步骤理解一遍。

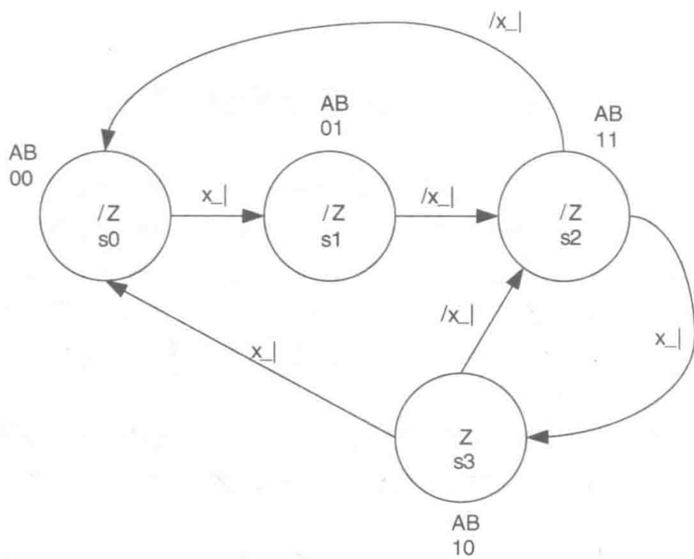


图 4.1 用于做对比的状态图

表 4.1 FSM 的当前和下一状态对应关系表

	第一列	第二列	第三列	第四列	第五列
	当前状态	下一状态	下一状态	下一状态	下一状态
	AB	AB	AB	DaDb	DaDb
		x = 0	x = 1	x = 0	x = 1
第一行	00	00	01	00	01
第二行	01	11	01	11	01
第三行	11	00	10	00	10
第四行	10	11	00	11	00

系统设计的下一步是写出 Da 和 Db 对应的公式。

找到表格中输入信号 x = 0 和 x = 1 时对应 Da 值为逻辑 1 的那一格。例如，Da = 1 时，第二行触发器 A 的输出从 0 变为 1；第三行的第一列和第三列分别表示触发器 A 的值原本为 1，下一状态也为 1，此时 x = 1 等。

当 AB = 01（第二行），x = 0 时，触发器 A 输出为高，此时的乘积项为/AB · /x。

当 AB = 01，x = 1（第二行第三列）时，触发器 A 被复位，此时的乘积项/AB · x 是不需要写入公式的。

当 AB = 10（第四行），x = 0 时，触发器 A 输出为高，乘积项 A/B · /x 为公式的组成部分之一。

当 AB = 11（第三行），x = 1 时，触发器 A 输出为高，乘积项 AB · x 是需

要的。

因此，Da 的表达式为 D · a = /AB · /x + A/B · /x + AB · x

公式没法简化。同理可以得出 D · b 的表达式：D · b = /A/B · x + /AB · /x + /AB · x + A/B · /x，这里 D · b 公式可以简化为 D · b = /A · x + /AB + A/B · /x

作为一个摩尔（Moore）状态机，输出信号 Z 的表达式为 Z = s3 = A/B。

现在用前 3 章介绍的方法来做同样的事情。

根据状态图，直接可以写出 D · a 和 D · b 的公式为

D · a = s1 · /x + s2 · x + s3 · /x = /AB · /x + AB · x + A/B · /x

D · b = s0 · x + s1 + s3 · /x = /A/B · x + /AB + A/B = /A · x + /AB + A/B · /x

结果和用常规方法得出的一模一样。

本书介绍的方法主要优势在于它不需要状态表来辅助得出触发器和输出信号的公式。尤其是当输入信号数量较多时（通常设计大规模 FSM 时面对的情况），表达当前状态和下一状态就会变得更加复杂，相比之下本书介绍的方法更加便捷。

4.2　处理未使用的状态

当我们设计状态图时，若其状态总数少于 2^n 个（n 表示二次状态变量的个数），必须考虑如何处理没有使用到的状态。图 4.2 给出一个例子。

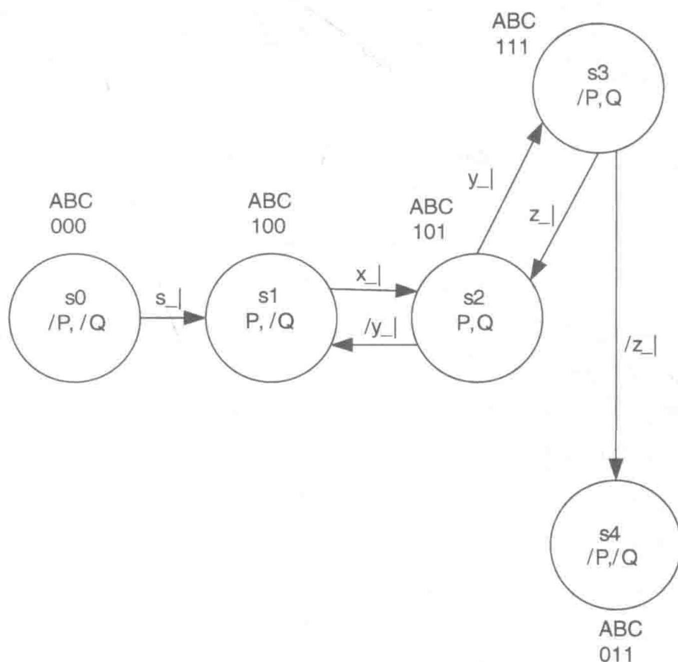


图 4.2　含有少于 2^3 个状态的状态图

二次状态变量和状态之间的关系表如下：

使用的状态	未使用的状态
s0 = 000	s5 = 010
s1 = 100	s6 = 110
s2 = 101	s7 = 001
s3 = 111	
s4 = 011	

D 触发器的公式分别为

$$A \cdot d = s0 \cdot s + s1 + s2 + s3 \cdot z$$

$$= /\cancel{A}/B/C \cdot s + A/B/\cancel{C} + A/B/\cancel{C} + A/\cancel{B}C \cdot z$$

其中用横线划去的表示运用了逻辑相邻定律和辅助定律进行简化的部分（参考本书附录 A)。最终结果为

$$A \cdot d = /B/C \cdot s + A/B + AC \cdot z$$

$$B \cdot d = s2 \cdot y + s3 \cdot /z + s4$$

$$= A/BC \cdot y + /\cancel{A}BC \cdot /z + /ABC$$

$$= A/BC \cdot y + BC \cdot /z + /ABC$$

$$C \cdot d = s1 \cdot x + s2 \cdot y + s3 + s4$$

$$= A/B/C \cdot x + A/\cancel{B}/C \cdot y + /\cancel{A}BC + /\cancel{A}BC$$

对于触发器 C，同样运用逻辑相邻定律和辅助定律将公式进行简化 $C \cdot d = A/B/C \cdot x + AC \cdot y + BC$。

输出公式为

$$P = s1 + s2 = A/B/C + A/BC$$

$$P = A/B$$

$$Q = s2 + s3 = A/BC + ABC$$

$$Q = A/BC + ABC = AC$$

如果状态机进入未使用的状态 s5（/AB/C)，则结果会变成 $A \cdot d = 0$，$B \cdot d = 0$ 以及 $C \cdot d = 0$。状态机会回到 s0。

如果状态机进入未使用的状态 s6（AB/C)，则会有 $A \cdot d = 0$，$B \cdot d = 0$ 以及 $C \cdot d = 0$，同样状态机会回到 s0。

如果状态机进入未使用的状态 s7（/A/BC)，则会有 $A \cdot d = 0$，$B \cdot d = 0$ 以及 $C \cdot d = 0$，状态机的下一个状态则又一次是 s0。

这表明用 D 触发器设计的 FSM 具有自我复位功能。

如果换作 T 触发器，FSM 则不会自动复位，因为触发器需要输入端 T = 1 才能触发，否则如果 T = 0 则维持在原本的状态。图 4.3 为能够让状态机返回 s0 的方法，显然，在使用 T 触发器时，每个触发器所对应的公式需要更多的乘积项。

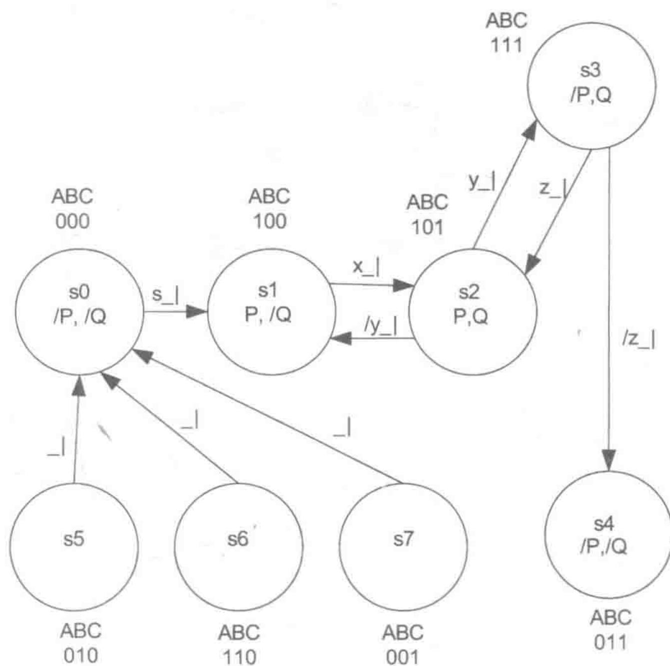


图 4.3　使用 T 触发器构建的状态机

一般情况下，如果状态机内部有多个 1 到 1 变换和少部分 1 到 0 变换以及 0 到 1 变换，使用 T 触发器可能会减少公式里乘积项的数量，即减少整个系统设计所需的逻辑单元。

如果系统内部只有少部分 1 到 1 变换，那么用 D 触发器所需要的资源更少。然而，由于 D 触发器的自动复位功能，总体而言它们在设计中能够体现更多的优势。

余下的章节，我们将介绍一些工程案例，并运用前 3 章介绍的方法。

4.3　信号高/低位指示系统

图 4.4 是系统的功能框图。图中 FSM 的任务是控制模 - 数转换器（ADC）以及监控模拟信号的转换结果，看其是否超出设定的上限值或者下限值。上限和下限的设定在系统中分别为输入信号 Hi - word 和 Lo - word，硬件上可以用双列直插式开关来实现。这里用的比较器是标准的 8 位比较器电路，类似于 7485 系列。这些在现有的 PLD 或者 FPGA 器件中都已经大规模集成了。

本系统的设计思路是，当模 - 数转换器（ADC）的输出端 A，超出了预设的上限值 Hi - word，信号 hi 将被置高。如果是低于下限值 Lo - word，则信号 lo 将被置高。ADC 接入系统的方式可以是单个器件，也可以是 PLD/FPGA 芯片内部集成的

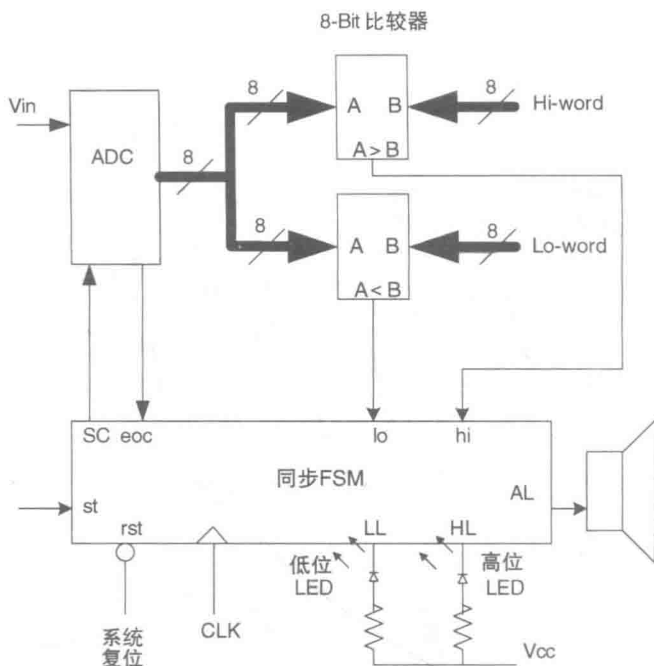


图 4.4 信号高/低指示系统框图

数字版本，然后在输出端外接一个 R/2R 的电阻网络。

系统在信号 st 被置高后即进入工作状态，在系统时钟的采样频率范围内，模 - 数转换操作会持续不断地进行，一旦出现 Hi - word 或者 Lo - word 的限定值被超越，相应的 LED 灯会被点亮，系统停止工作。此时按下复位按钮，系统将回到初始态。注意这里的报警器在模 - 数转换输出值等于上限值 Hi - word 或者下限值 Lo - word 时不会发出告警声。

根据上述功能描述，可以开始设计系统状态图。控制模 - 数转换器的方法可以参照第 2 章所讲述的相关内容。

由组合逻辑构成的两个数字比较器，将根据模 - 数转换器的输出电平给出相应的比较结果。当模 - 数转换器的输出等于或小于 Hi - word，但是大于 Lo - word 时，表明模 - 数转换器的输出在两个限定值中间，此时信号 hi 和 lo 将均为逻辑 0。当模 - 数转换器的输出大于上限值 Hi - word，信号 hi 将被置高，系统会拉响警报器，并且点亮 High 指示灯。当模 - 数转换器的结果小于下限值 Lo - word，信号 lo 被置高，警报器也会响，Low 指示灯点亮。

图 4.5 是系统对应的状态机框图。系统在状态 s0 时处于上电或者复位状态，并等待启动信号 st 被置高。随后信号 SC 被置高，表明模 - 数转换器开始工作，将模拟信号转换为数字信号。之后系统进入状态 s2，并检测模 - 数转换器的输出，

无论是上限值还是下限值一旦被超过，状态机会进入 s3。如果转换结果处于中间范围，状态机将回到 s1，开始新的一轮模 - 数转换的循环。

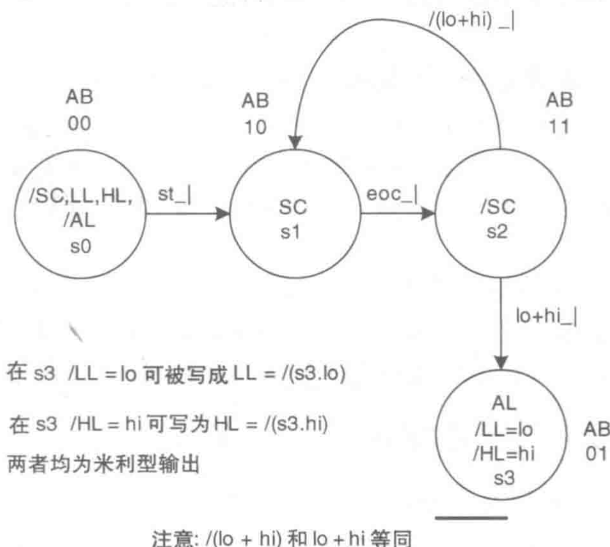


图 4.5　系统状态图

大家注意观察状态图中的状态 s2 处出现了两路分支，信号 lo 和 hi 作为或门的两个输入端，其结果取反后为/(lo + hi)。此外，如果这时将 De Morgan 法则应用到表达式/(lo + hi)，会得到/lo · /hi，表明从状态 s2 回到状态 s1 的条件是两个信号 lo 和 hi 都必须为低（逻辑 0）。

大家再看一下状态 s3，输出信号 HL 和 LL 的状态，分别受比较器的输出信号 hi 和 lo 控制，因此，当 hi = 1 时指示灯 HL 在 s3 被点亮，同理 lo = 1 时 LL 指示灯亮。

/HL = hi 表示 HL 为低有效信号，所以输出信号 HL 的表达式应为 HL = /(s3 · hi)，表明当 hi = 1 时 HL 为逻辑 0，但是有一个前提条件，就是系统必须处于状态 s3。这属于一种米利（Mealy）型低有效输出，在第 3 章中已经介绍过了。

同样地，LL = /(s3 · lo)。

想要记住低有效的最简便的方法，就是在状态 s3 时想到/HL = hi 这样的关系，而不是上面的公式。就像图 4.5 里所展示的那样，在状态 s3 的圆圈里，写出不包含 s3 的输出信号表达式。

现在将状态 s3 用它的二次状态变量来替代，即 AB = 01，此时两个指示灯的表达式可以重新写为 HL = /(s3 · hi) = /(/AB · hi)，LL = /(s3 · lo) = /(/AB · lo)。

新的表达式代表了 2 个 3 路输入的与非门。要记住低有效信号是取反的（参考第 3 章有关内容）。

因此，从 HL 的公式可以看出，当系统处于状态 s3 时，A = 0（/A = 1），B =

1，如果此时 hi = 1，与非门的输出则为逻辑 0，正好符合点亮 LED 指示灯的要求（低有效）。

看完米利（Mealy）型输出的一些具体特性之后，可以写出触发器 A 和 B 的表达式了。

使用第 3 章介绍的方法，很快可以得出：A · d = s0 · st + s1 + s2 · /(lo + hi) = /A/B · st + A/B + AB · /hi · /lo。

使用辅助规则，A · d 表达式可以简化为 A · d = /B · st + A/B + A · /lo · /hi。

触发器 B 的表达式为 B · d = s1 · eoc + s2 · (lo + hi) + s3 = A/B · eoc + AB · lo + AB · hi + /AB。

再次用辅助规则简化表达式 B · d = A/B · eoc + B · lo + B · hi + /AB。

余下的输出为摩尔型，SC = s1 = A/B，AL = s3 = /AB。

下一步是编写 Verilog HDL 代码构建 FSM 和比较器。

4.3.1 使用测试平台测试 FSM

图 4.6 是系统仿真结果，它是用测试软件并根据 Verilog HDL 代码生成的。软

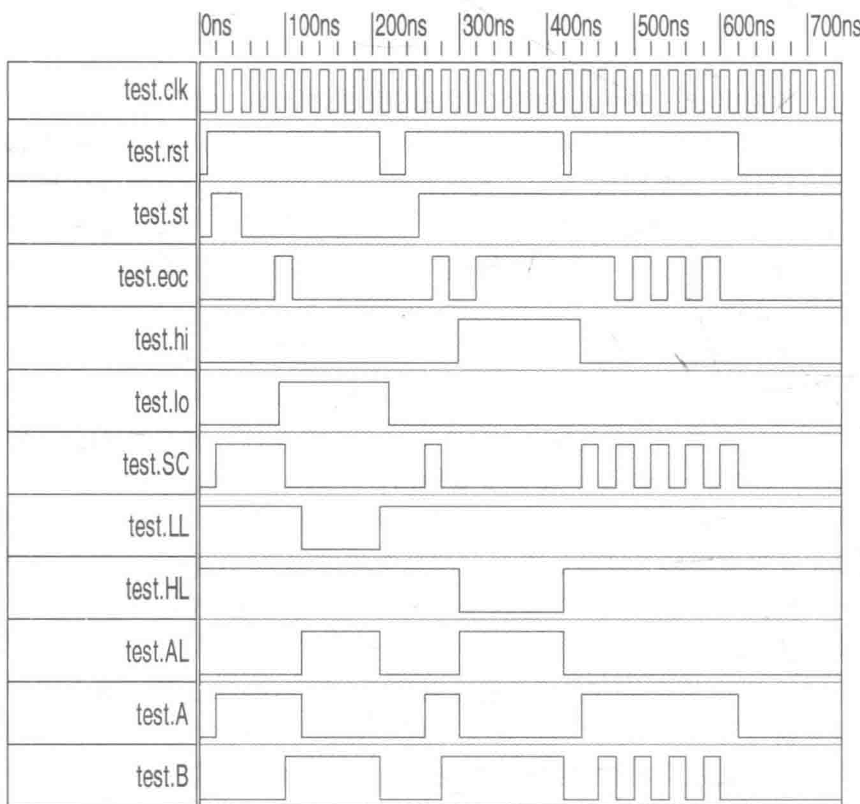


图 4.6 FSM 仿真测试

件对所有信号和 FSM 的各个状态均进行了仿真测试。整个测试过程是首先让系统以状态 s0→s1→s2→s3 的顺序开始，然后模 - 数转换器的输出值超出了下限且 FSM 停在了 s3（A = 0，B = 1），直到复位信号（rst = 0）被启动。然后系统再次进入另一个循环，顺序为 s0→s1→s2→s1→s2→s1→s2→s1→s2→s0，表示测试值没有超过上下限，然后需要等待另一个 rst = 0 的复位信号，才能让 FSM 回到 s0。经过这两个过程，FSM 的功能测试才算完整。

4.4　简易波形发生器

某些时候，可能需要产生一个波形去测试生产线上的产品。一种办法是使用晶体振荡器来达到这个目的，但是如果所需要的波形不是纯粹的正弦波、方波、锯齿波或者三角波，基于晶振的方案会显得比较繁杂。还有一种方法是用微控制器和数 - 模转换器（DAC）的组合，来产生多种波形。每一种波形的参数都可以存在只读存储单元（ROM）中，然后使用微控制器来读取。然而这样的方法显得有些多余，而且微控制器对采样频率也有限制。而使用带时钟信号的 FSM，来完成这个操作就比较灵活。采样频率由输入时钟来控制，可以通过 PLD 或者 FPGA 器件来分频或者倍频，各种波形的参数还是存在只读存储单元（ROM）里，只不过是用 FSM 来控制它们的读取。

图 4.7 给出了系统的框图，系统在 st 信号被拉高后便开始工作。每种波形的数字格式都存放在存储单元的各个地址空间里，当它们按顺序被逐个访问时，波形文件会被送往数 - 模转换器，转成模拟信号输出。当某一种波形对应的所有地址空间都被访问到的情况下，地址计数器会归零，从头开始计数。

st 信号被拉低会暂停系统运行。实际的采样率，即波形的周期是可以在完成状态图的设计之后计算出来。数 - 模转换器的结果需要在输出之前滤除高频分量，如果采样频率高出波形实际频率过多，可以使用一阶低通滤波器来去除噪声（通常情况下，采样频率满足香农采样理论即可）。

现在可以设计状态图了。不过观察系统框图，有些要点需要提前考虑：

1）地址计数器必须在最初时复位，确保其指向地址空间的起始位。系统需要在 st 信号被激活（置 1）之前确保处于状态 s0。

2）存储单元随后需要被使能、选中，允许被访问，紧接着存储单元里的数据将被送往数据锁存器的输入端口。数据被锁存后即可送往数 - 模转换器。

3）此刻，地址计数器需要通过累加来指向下一个存储空间，随后第二步所做的事情会重复一遍，只要 st 信号始终处于激活状态，系统就会一直循环下去。

在本系统中，不需要担心地址计数器指向最后一个存储器空间之后的动向，因为这里的设计方案是计数器会自动归零，从头开始。这也表明存储空间里存放的是一个周期的完整的波形数据，因此只要计数器不停，在输出端将看到连续不断的完

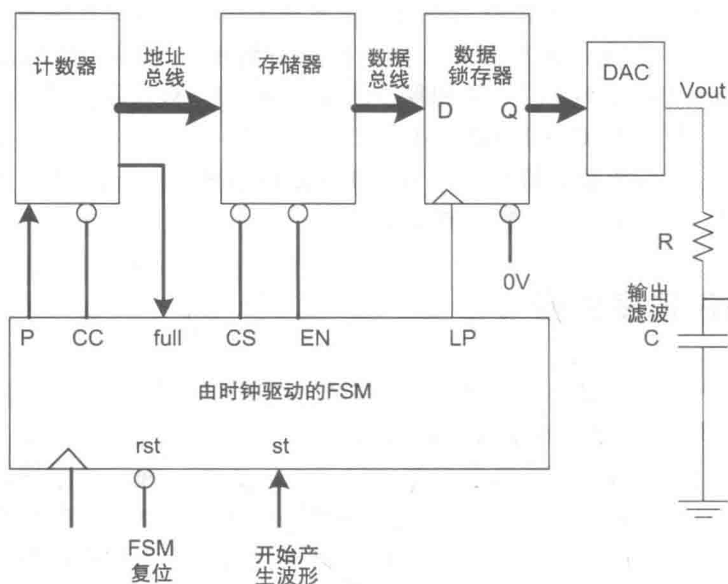


图 4.7 简易波形发生器系统框图

整的波形。通过在系统外部集成一些外围电路，可以将上述完整输出波形的要求得到完善，但是这里不再详述。

根据上述系统功能描述，便可生成状态图。

图 4.8 中所示的状态图和之前描述的系统功能一致。在状态 s3，输出信号 P 是一个米利（Mealy）型输出。P 是只有在状态 s3，且时钟的信号为低时才有效（置 1）。这样确保了地址计数器在地址使能信号 EN 被释放（拉高）后才开始计数（P 的上升沿）。因此，从存储器读取的数据，在切换地址空间时处于三态状态。使用数据锁存器确保了数 - 模转换器的输入数据的完整性。关于信号 P 的另一种方案是在状态 s3 和 s1 之间加入一个冗余状态，条件是 P = 1，这样可以避免系统在输出信号 P 端的一些不稳定性（具体参考本书第 1 章）。

现在写出各个部分的公式：

$$
\begin{aligned}
A \cdot d &= s0 \cdot st + s1 + s3 \\
&= /A/B \cdot st + A/B + /AB \\
&= /B \cdot st + A/B + /AB \\
B \cdot d &= s1 + s2 \\
&= A/B + AB \\
&= A
\end{aligned}
$$

输出表达式为

CC = /s0 = /(/A/B)，低有效输出；

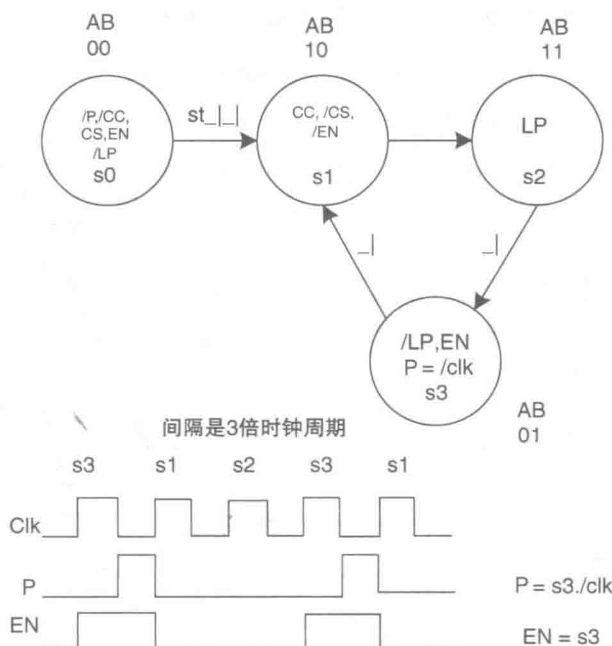


图 4.8　简易波形发生器的完整状态图

CS = s0 = /A/B，也是低有效输出，但只在状态 s0 有效；

LP = s2 = AB；

EN = s0 + s3 = /A，只在这 2 个状态有效；

P = s3 · /clk = /AB · /clk，米利型输出，受时钟信号控制。

用 Verilog HDL 语言可以很方便地表达上述公式，只是表达方式要遵循 Verilog HDL 的语法规则。与门用 & 表示，或门是 | ，非门是 ~ ，而异或则是^。

用一个 assign 语句可以将这些公式全部包括进来，即

```
assign
A · d = ~B&st | A& ~B |  ~A&B;
B · d = A;
CC = ~ ( ~A& ~B);
CS = ~A& ~B;
LP = A&B;
EN = ~A;
P = ~A&B& ~clk;
```

在本书附录 C 里详细说明了如何将 Verilog 文件导入系统来仿真状态机。在第 6 和第 8 章也将会有更详细的说明。

4.4.1 采样频率和每种波形的采样个数

从图 4.8 可以明显看出，系统每读取一个存储空间，需要经过 3 个状态来完成，因此采样周期是时钟周期的 3 倍。

因此，如果采样频率为 300kHz，那么采样时钟最起码得是 900kHz。如果考虑到系统的周密性，可以在循环中加入一个冗余状态，这样让时钟周期为采样频率的 4 倍比较好。

存储空间大小是随机的，且影响到地址计数器的规模。例如内存大小为 1KB，那么地址计数器所需要的触发器数量为触发器数量 = ln （1024）/ln （2）= 10。

图 4.9 给出了仿真结果。

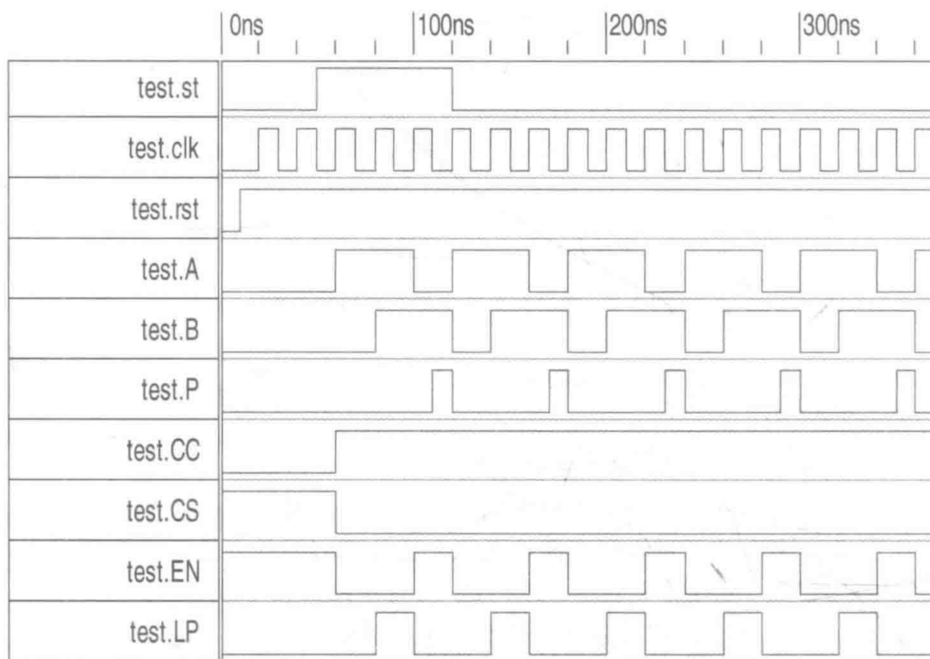


图 4.9 波形发生器仿真图

4.5 骰子游戏

本系统包括 7 个 LED 指示灯、输入信号 p 和一个时钟。图 4.10 为系统框图，输入信号 p 为开关，输入时钟可以由一个简单的晶振电路构成，系统带有一片频率为 100Hz 的 555 计时器芯片，目的是增加指示灯的闪烁效果。

LED 指示灯的界面分布如图 4.11 所示，用来模拟真实情况。系统所用的 LED 是低功耗正向 2mA 发光二极管。这样当电源端为 5V 时，限流电阻为 1800Ω。同时

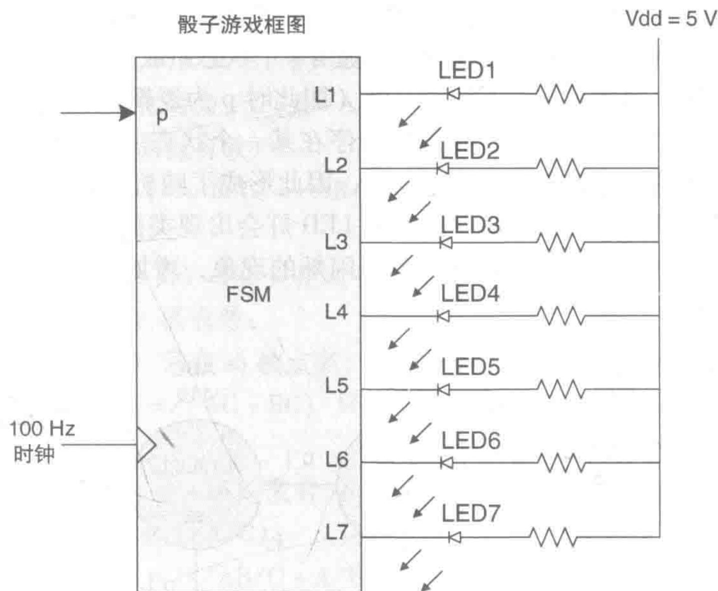


图 4.10　骰子游戏系统框图

FSM 对应 LED 的输出端设置为开漏型。图中包含 6 种指示灯点亮的情况，即掷骰子的 6 种可能性，但所有灯均不亮的情况不在其中。

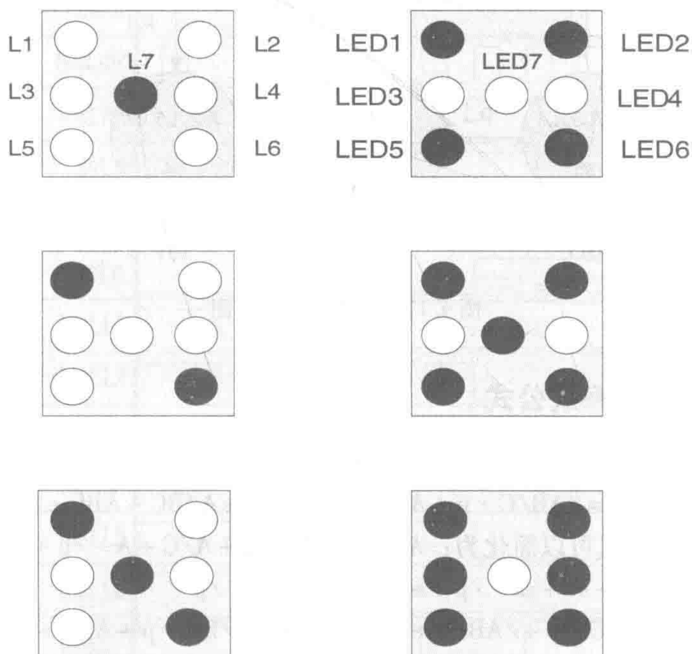


图 4.11　LED 指示灯界面

状态机的设计比较简单直观，所要做的事情就是依次显示每一个数字，但是显示速度要超过玩游戏的人的反应速度。系统由 7 个状态组成，每一个都显示固定的数字格式。按钮 p 每次被按下不动的时候（即此时 p 为逻辑 1），状态机则开始不停地切换状态。当按钮被释放，状态机会停在某一个状态。由于系统时钟频率较高，操作者是无法跟上状态切换的速度的，因此形成了随机性。但需要注意的是，如果时钟频率过高，按下 p 之后，所有的 LED 灯会出现类似一直常亮的现象。因此合适的时钟频率会形成指示灯界面不停闪烁的现象，增加游戏的乐趣。图 4.12 是系统的状态图。

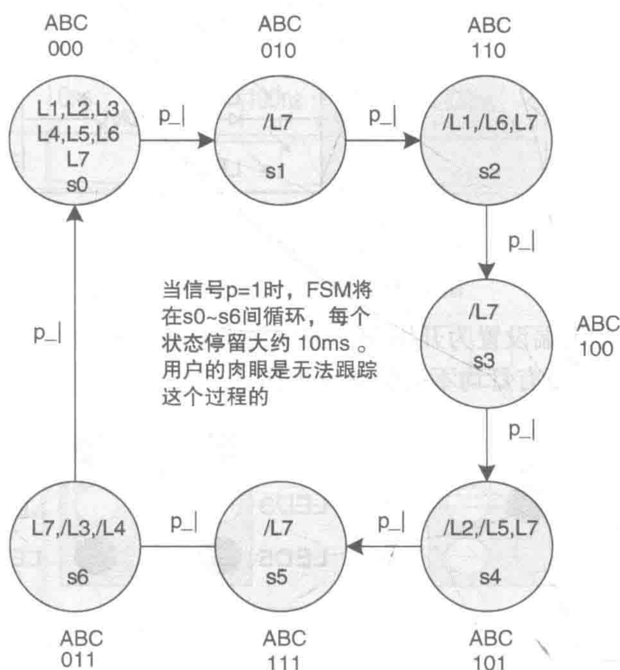


图 4.12　骰子游戏状态图

4.5.1　骰子游戏系统公式

$$A \cdot d = s1 \cdot p + s2 + s3 + s4 + s5 \cdot /p$$

$$= /AB/C \cdot p + AB/C + A/B/C + A/BC + ABC \cdot /p$$

触发器 A 的公式可以简化为：$A \cdot d = B/C \cdot p + A/C + A \cdot /p + A/B$

$$B \cdot d = s0 \cdot p + s1 + s2 \cdot /p + s4 \cdot p + s5 + s6 \cdot /p$$

$$= /A/B/C \cdot p + /AB/C + AB/C \cdot /p + A/BC \cdot p + ABC + /ABC \cdot /p$$

触发器 B 的公式简化后为：$B \cdot d = /A/C \cdot p + /AB/C + AC \cdot p + ABC + B \cdot /p$

$$C \cdot d = s3 \cdot p + s4 + s5 + s6 \cdot /p$$
$$= A/B/C \cdot p + A/BC + ABC + /ABC \cdot /p$$

触发器 C 的公式简化后为：$C \cdot d = A/B \cdot p + BC \cdot /p + AC$

输出公式（LED 均为低有效）：

L1 =（s0 + s1）=（/A/B/C +/AB/C）=/A/C 只在 s0 和 s1 两个状态是熄灭的；

L2 =（s0 + s1 + s2 + s3）=/C 只在前 4 个状态是熄灭的；

L3 =/s6 =/(/ABC）低有效；

L4 =/s6 =/(/ABC）只在 s6 被点亮，因此表达方法上直接取反；

L5 =/(s4 + s5 + s6）=/(AC + BC）只有在 S4 ~ S6 三个状态被点亮，因此逻辑或之后取反；

L6 =/(s2 + s3 + s4 + s5 + s6）或者为（s0 + s1），因为 L6 只在 s0 或者 s1 状态是熄灭的，所以可以写成（/A/C）；

L7 =/(s1 + s3 + s5）=/(/AB/C + A/B/C + ABC）。

图 4. 13 为系统仿真图，二次状态变量 a、b 和 c 的状态变化也在其中，所有 7 个 LED 指示灯的点亮状态也和图 4. 11 给出的一致。

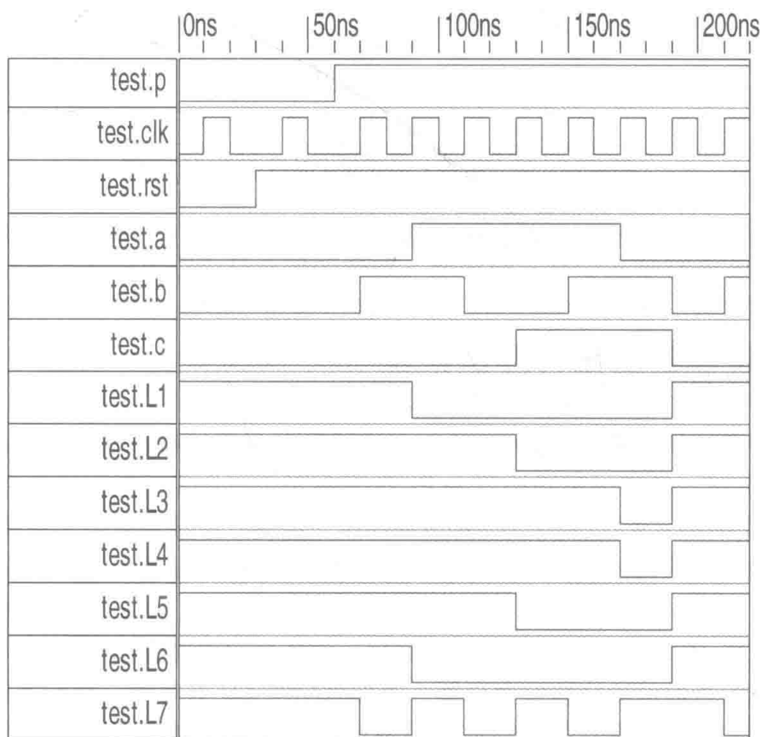


图 4. 13　骰子游戏仿真图

图 4.14 模拟了用户将按钮 p 按下又松开的情况，当按钮被松开时，FSM 停在了 s3，然后在 p 被再次按下时又继续工作。

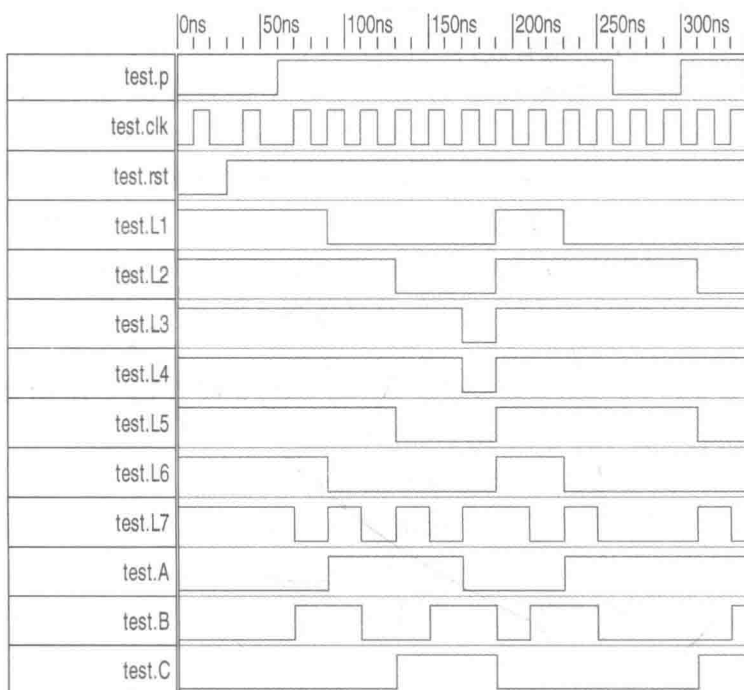


图 4.14 输入信号 p 出现变化时骰子游戏仿真图

注意到两个仿真图中时钟的单位都是 ns，实际情况下，时钟周期可以调慢到 10ms 左右。

4.6 二进制数据串行发送系统

本节将介绍如何将 4 位（4 - bit）二进制数据，通过移位寄存器以串行的方式向外界发送。

图 4.15 给出了系统框图，FSM 用来控制二进制计数器和并行加载移位寄存器。关于这两个器件的具体设计方案，大家可以参考附录 B 用 Verilog HDL 语言建模的方法。

系统在输入信号 st 被拉高后开始运行。FSM 将计数器的复位信号释放，然后将计数器里的当前值送往移位寄存器。当加载信号 LD 有效时，移位寄存器将根据时钟控制的波特率，将计数器当前的值通过输出（TX）端发送出去。当移位寄存器中没有数据时，对应的 RE 信号将置高，并通知 FSM 寄存器为空。通过观察 RE 信号，可以得知计数器的值是不是全部发送完毕。当所有数据发送完毕后，信号

done 被置 1，并通过检测模块（一个与门）反馈给状态机。如果还有数据未发送，那么系统将继续把数据送往移位寄存器。此后，系统将停止工作，并等待 st 信号回到其释放状态，随后系统回到状态 s0。

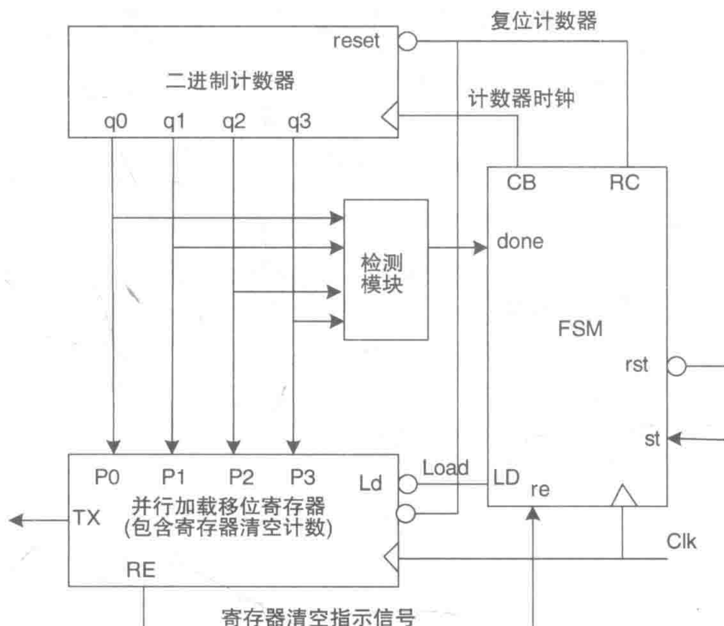


图 4.15　二进制数据串行发送系统框图

根据描述，图 4.16 为对应的系统状态图。状态图的设计本身没有问题，但是二次状态变量不符合单位距离编码的格式。如果加入一个冗余状态 s7，那么在 s6 和 s0 之间的二次状态变量 A、B 和 C 就可以形成单位距离编码的格式。注意，在图 4.17 中，除了信号 RC，状态 s7 和 s0 的输出是相同的，尽管在图中没有明示。从状态 s5 到状态 s1 的转换也不是单位距离编码格式，这也会导致输出端存在一些潜在的隐患，如果在状态 s5 和 s1 之间加入冗余状态，形成单位距离编码格式或许是一种解决方案。请读者自己尝试解决 s5 和 s1 之间的输出隐患，但是有一点必须提醒大家的是，由于状态 s7 的存在，更多的冗余状态则需要多加一位二次状态变量（即 D 触发器），因为当前 3 个二次状态变量最多只能容纳 8 个状态。

根据图 4.17，可以写出每个触发器对应的公式

$$A \cdot d = s1 + s2 + s3 + s4$$

$$= /AB/C + AB/C + A/B/C + A/BC$$

简化后为：$A \cdot d = B/C + A/B$

$$B \cdot d = s0 \cdot st + s1 + s4 \cdot re + s5 + s6 \cdot st$$

$$= /A/B/C \cdot st + /AB/C + A/BC \cdot re + ABC + /ABC \cdot st$$

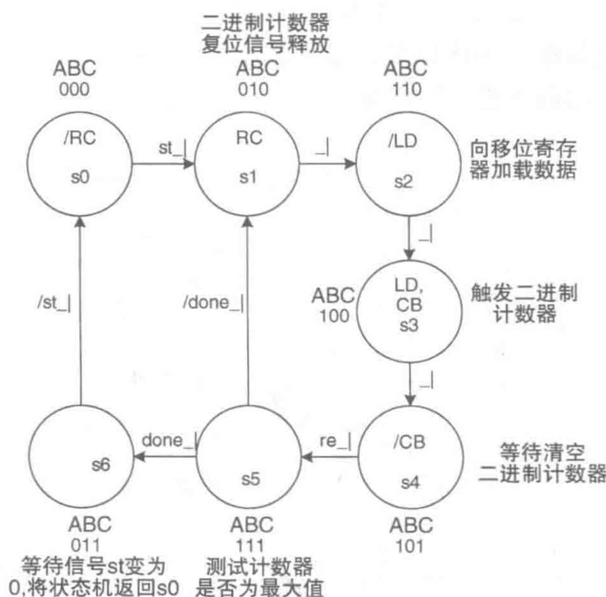


图 4.16 二进制码串行发送状态图

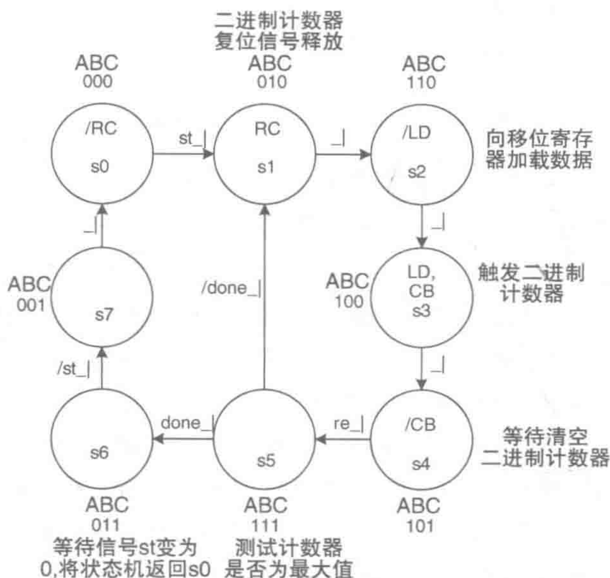


图 4.17 加入冗余状态 s7 形成二次状态变量的单位距离编码格式

简化后为：B · d = /A/C · st + /AB/C + AC · re + BC · st + ABC

C · d = s3 + s4 + s5 · done + s6

= A/B/C + A/BC + ABC · done + /ABC

简化后为：C · d = A/B + BC · done + /ABC

输出信号（摩尔型）公式为：

RC = /s0 = /(/A/B/C) 低有效

LD = /s2 = /(AB/C)

CB = s3 = A/B/C 高有效

系统仿真如图 4. 18 所示，状态机的运行情况是通过将二次状态变量 A、B 和 C 的值，纳入仿真系统来完成的。它们值的变化可以参照图 4. 17。

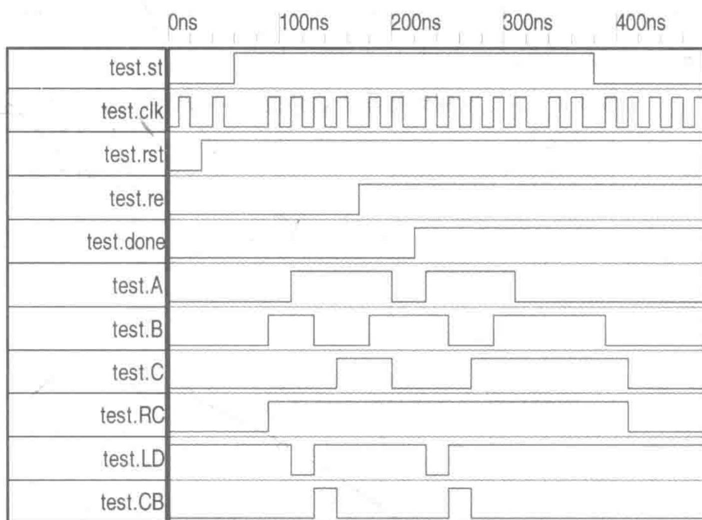


图 4. 18　二进制数据串行发送器仿真图

4. 6. 1　图 4. 15 移位寄存器里的 RE 计数单元

图 4. 15 中的移位寄存器有一个输出信号 RE，它是寄存器被清空的指示信号。使用一个 4 位计数器可以实现其功能，计数器在输入信号 Load 被释放（拉高）时开始工作。计数器的工作时钟和移位寄存器的时钟是一样的，当它达到最大值 1000 时，可以使用最高位作为 RE 信号。

表 4. 2 表述了其工作原理。

从表 4. 2 中可以看到，计数器在第 8 个时钟脉冲将最高位 D 置 1，此时将这一位用作寄存器清空指示信号 RE。当数据全部发送完毕后，FSM 回到状态 s0，RC 信号随后将再次被拉低，将二进制计数器和 RE 计数器全部复位。除了上述方法，二进制计数器还可以再外挂一个触发器用作 RE 指示信号。

大家可以参考附录 B 来设计 RE 计数器的公式。如果用 T 触发器来设计，那么公式可以写成：

A · t = 1

B · t = A

C · t = AB

D · t = ABC

RE = D

在附录 B 中，会告诉大家如何基于这几个布尔代数公式，直接用 Verilog HDL（或可以用其他硬件描述语言代替）来设计 4 位计数器。

在附录 B 里还给出了用 T 触发器设计同步计数器的方法。当然根据需要，计数器也可以被设计成异步的。

表 4.2 用二进制计数器表达移位寄存器清空标志

二进制计数器					
RE					
D	C	B	A	计数值	
0	0	0	0	0	
0	0	0	1	1	
0	0	1	0	2	
0	0	1	1	3	
0	1	0	0	4	
0	1	0	1	5	
0	1	1	0	6	
0	1	1	1	7	
1	0	0	0	8	当 D = 1 时表示移位寄存
1	0	0	1	9	器清空，且 D 保持逻辑 1

同样在附录 B 里还收录了一个用 D 触发器设计的并行加载移位寄存器，这里将其公式从附录 B 里摘抄如下：

Q0 · d = din · ld + p0 · /ld （B. 7）

Q1 · d = q0 · ld + p1 · /ld （B. 8）

Q2 · d = q1 · ld + p2 · /ld （B. 9）

Q3 · d = q2 · ld + p3 · /ld （B. 10）

Sft _ clk = clk · ld （B. 11）

图 4. 19 为基于上述公式设计的 4 位（4 – bit）并行加载移位寄存器的电路图。

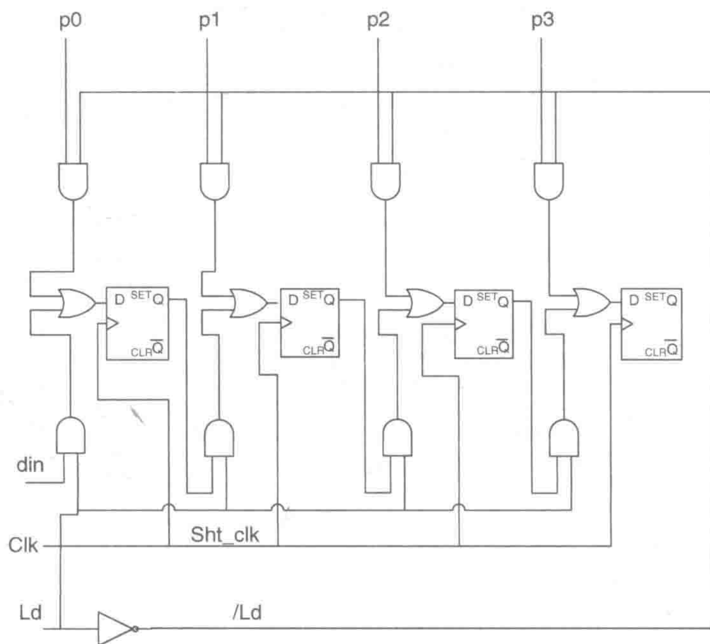


图 4.19　4 位并行加载移位寄存器电路图

4.7　串行异步接收系统

通常数字电路系统需要带有串行通信的收发功能。尽管市场上由很多串行通信的器件，但学会在 FPGA 芯片里内嵌自己设计的串行通信模块还是很实用的。这样好处在于数据传送的波特率和协议可以由设计者自己规定，同时也能够很好地控制整个系统。

在本节中，串行数据输入被接入一个带有起始位（st）和停止位（sp）的异步的数据包传送协议。这种协议提供了一种可以识别数据包如何到达端口的方法。并且不对数据的到达时刻产生任何限制，且速率是可调的（通过波特率控制）。

接收数据的关键是必须确保移位寄存器在某一时刻接收到相应的数据位。为了达到这个目的，FSM 会在接收数据的这段时间里，为移位寄存器专门产生一个时钟 RXCK，来进行同步数据的接收。

RXCK 时钟的脉冲在图 4.20 里用箭头表示，和主时钟的关系是每隔 3 个主时钟脉冲产生一个 RXCK 时钟脉冲，即主时钟频率是 RXCK 时钟频率的 4 倍。要注意的是，产生 RXCK 时钟之前，FSM 需要检测到数据包的起始位，协议规定数据输入端出现 1 到 0 的转换时，即代表当前一位数据是数据包的起始位。

图 4.21 是异步串行接收系统框图，FSM 的任务是产生移位寄存器的时钟，并

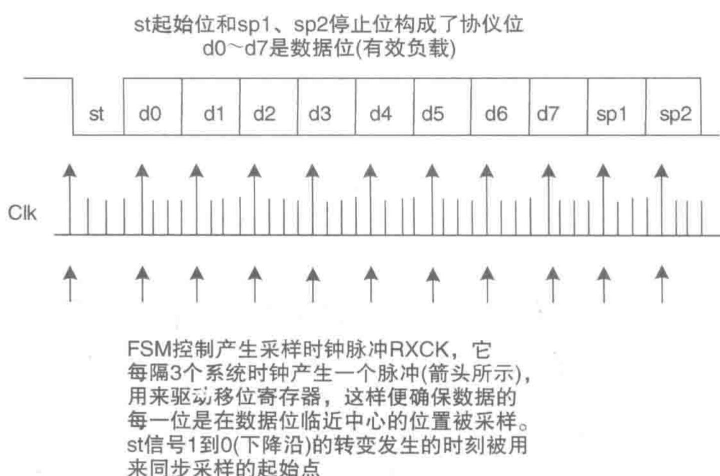


图 4.20　异步串行接收系统数据协议

且控制异步串行数据的接收。4 位计数器用来检测收到的 11 位（11 - bit）数据包，同时计数器还产生一个移位寄存器满载的 rxf 信号，通知 FSM 接收到了一包完整的数据。数据锁存器的工作是把从移位寄存器送来的数据锁存，并发送到外部其他（控制本系统的）器件或系统。

FSM 需要等待一个开始信号（信号 st 从 1 变为 0），且此时接收到的第一位数据正好进入移位寄存器。此后一旦有数据，系统便将其送入移位寄存器。如果最终的停止位出现错误，FSM 会发出一个错误信号 ERR。注意，本系统中每一数据包的开始位和两个结束位的检测是通过一个三路输入的与门来实现的，与门的输出为信号 ed，移位寄存器输入信号 rx 在内部由一个上拉电阻将其保持在高位（逻辑 1），这样系统就能检测到开始信号 st（低有效）被激活。信号 ack 帮助使用本系统的外部器件或电路，及时检测到错误的产生（没有错误代表数据包接收正常）。数据锁存器里的值是外部器件或电路需要读取的真正数据（8 位）。

信号 CDC 用来清空移位寄存器并将信号 st 设为逻辑 1，移位寄存器中代表起始位的触发器的输出端在接收数据之前需要预设为逻辑 1，这样当数据包的开始位到来时才能将其下拉到低电平。

信号 en 是整个异步接收系统的使能信号。这个信号的主要作用是让系统在既定的时刻根据系统时钟产生移位寄存器的数据接收时钟（RXCK）。

状态图如图 4.22 所示。图中 FSM 等待信号 en 被拉高以及开始信号 st 被拉低，然后系统经过状态 s1、s2 以及 s4 和 s5，将数据包的起始位存入移位寄存器。这几个状态所做的事情是确保系统检测到数据包的起始位，且在正确的时刻将其存入移位寄存器。在状态 s5，数据包的起始位是被时钟 RXCK 的脉冲送入移位寄存器的，然后系统进入状态 s6，将 RXCK 拉低，随后开始另一个循环，涉及的状态分别是

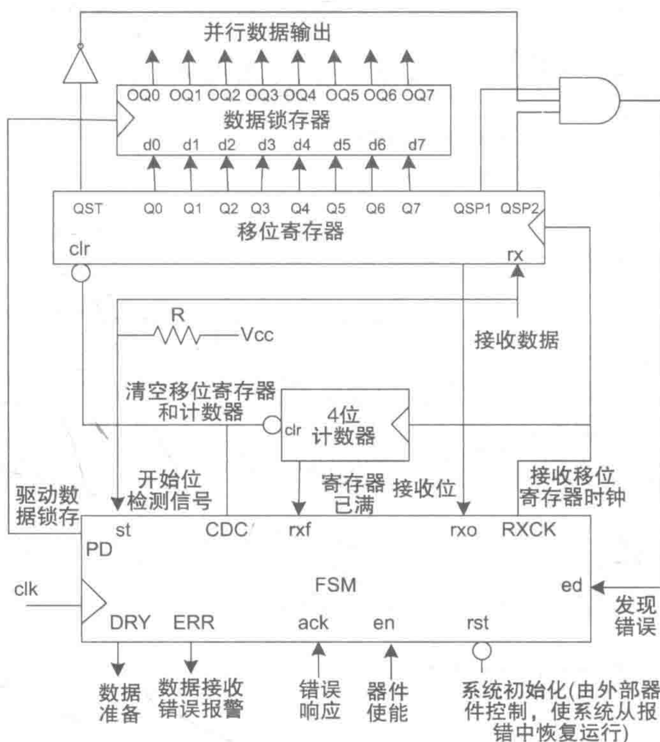


图 4.21　串行异步接收系统框图

s5、s6、s7 和 s8。

这几个状态的循环机制确保移位寄存器的时钟（RXCK）脉冲对准每一位数据的中央，在 11 位数据全部被送入移位寄存器之后，4 位计数器将发出信号 rxf，指示寄存器已经存满。然后 FSM 将进入状态 s9，开始检测起始位和停止位是否正确发送（此时信号 ed 应该为逻辑 1）。如果 ed = 0，FSM 将进入状态 s10，发出报错信号。

外围电路接收到报错后，可以将整个异步接收系统复位，重新开始。如果没有报错，FSM 会进入状态 s11，去锁存移位寄存器里保存的数据（OQ0 ~ OQ7），方便外部读取。同时，信号 DRY 用于提示外围电路数据已经处于可被读取的状态，且外围电路会通过信号 ack 反馈收到提示。当信号 ack 被拉低后（状态 s12），FSM 随后可以回到状态 s0 等待下一包数据。因此这里信号 DRY 和 ack 是 FSM 和外围电路的通信握手信号。

状态机只要一直处于工作状态，使能信号 en 会常高，直到所有数据接收完毕。

大家注意到状态图里没有状态 s3，这是由于在系统设计过程中，状态 s3 被移出系统。因为它会引起一个不必要的错误，而使得 s3 成为整个系统的累赘。而状态图设计本身应该是一个迭代的过程。

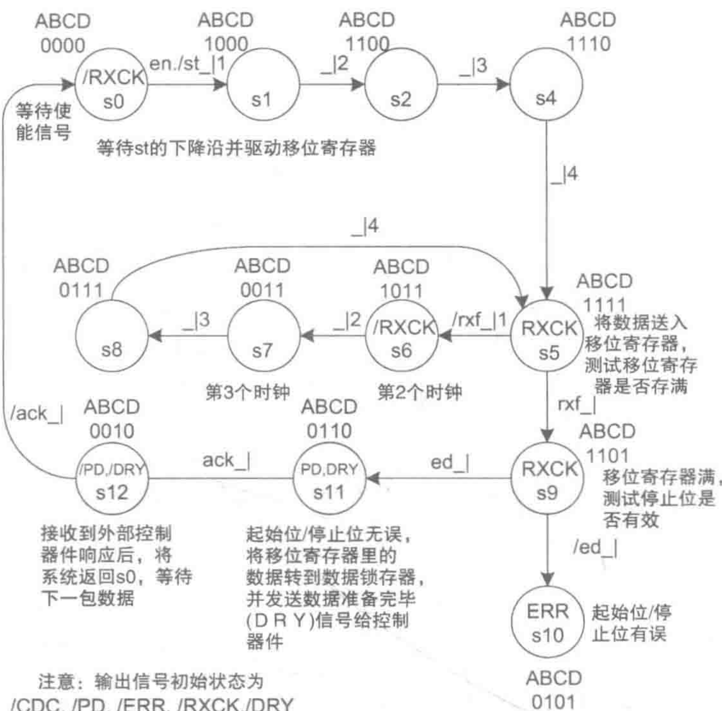


图 4.22　串行异步接收系统状态图

4.7.1　FSM 公式

A · d = s0 · en · /st + s1 + s2 + s4 + s5 + s8

B · d = s1 + s2 + s4 + s5 · rxf + s7 + s8 + s9 + s10 + s11 · /ack

C · d = s2 + s4 + s5 · /rxf + s6 + s7 + s8 + s9 · ed + s11 + s12 · ack

D · d = s4 + s5 + s6 + s7 + s8 + s9 · /ed + s10

RXCK = s5 = ABCD

PD = dry = s11 = /ABC/D

ERR = s10 = /AB/CD

读者可以自行将带有二次状态变量 A、B、C 和 D 的公式补充完整。

本系统的仿真和其他各个模块的设计方案，大家可以参考附录 B 进一步学习。

4.8　加入奇偶校验的串行接收系统

4.7 节介绍的数据接收系统还可以做一些优化，例如将第一位停止位 sp1 改为奇偶校验位。运用组合逻辑电路构建的奇偶校验功能，对数据包里的每一位数据既

可以进行奇校验，也可以进行偶校验。这需要一组异或门将数据包里的 11 位数据按位做异或运算。

例如，如果需要奇校验，那么发送数据时就要产生一个奇校验输出位 OP：OP = b0^b1^b2^b3^b4^b5^b6^b7^b8^b9^b10。

或者将起始位和停止位加入进去：OP_{n+1} = st^ b0^b1^b2^b3^b4^b5^b6^b7^b8^b9^b10^OP_n^sp

这个输出位在 FSM 端进行检测时，其结果必须为逻辑 1。如果是逻辑 0，则意味着数据包里至少有一位数据出错。

偶校验输出的产生方法就是奇校验的取反：EP_n = /OP_n。

运行奇偶校验功能时，在产生 OP_n信号的异或门之后再串联两个异或门，并将 OP_n作为它们的输入端，接收端所收到的最终校验结果被命名为 P。

参与校验的数据 d0，d1，…，d7 是移位寄存器每一次存满之后的输出。

4.8.1　整合奇偶校验

本节将一个奇偶校验模块的输入端和移位寄存器的输出端相连，校验模块的输出 OP_n和 OP_{n+1}作为比较器的输入，其结果作为校验的最终结果被送往 FSM，如图 4.23 所示。

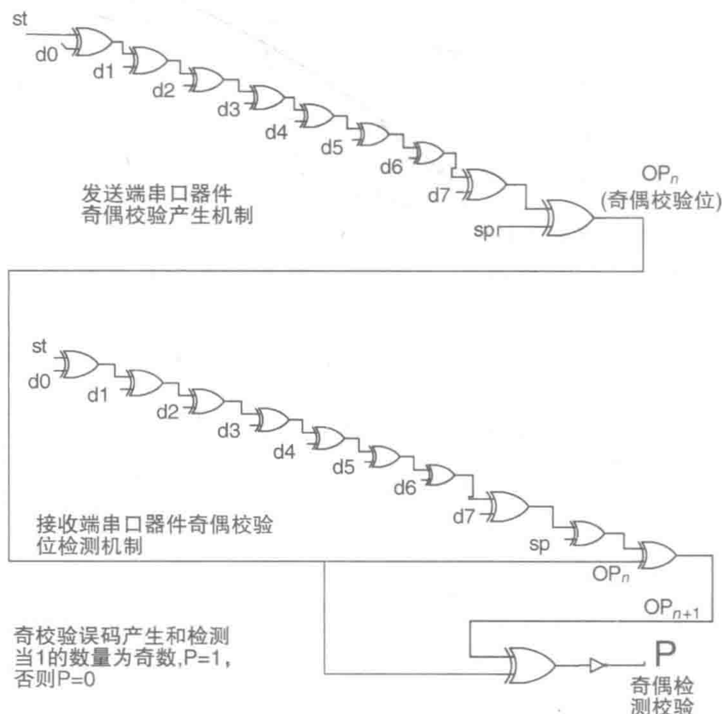


图 4.23　奇偶校验位产生和检测原理示意图

图 4.24 是加入了奇偶校验的数据传输协议示意图，图中校验位 OP_n 替代了原本的停止位 sp1。

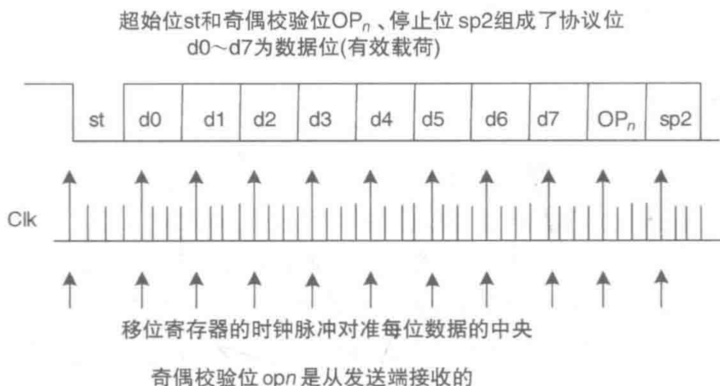


图 4.24 带奇偶校验的数据传输协议

图 4.25 是加入奇偶校验模块的系统框图，此刻系统可以检测移位寄存器停止位和校验位出现的错误，但这里没有测试起始位（不过可以根据需要加入）。

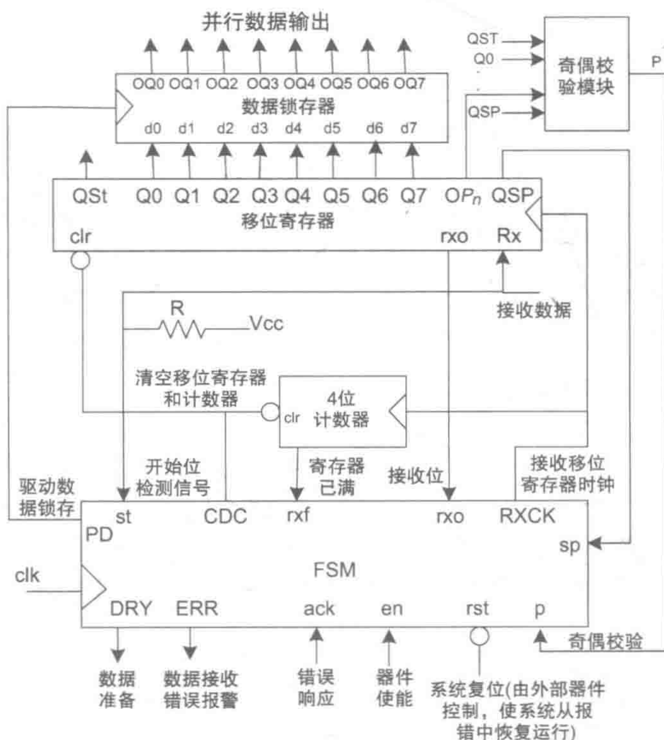


图 4.25 带奇偶校验的系统框图

图 4.26 是带有校验功能的状态图，本系统采用的是奇校验，校验输入 OP_{n+1} 必须和生成的校验位 OP_n 进行比较。如果比较结果相同，则代表数据没有错误。比较器可以用同或门来实现，如果 $OP_n = OP_{n+1}$，比较器输出 P 为逻辑 1，否则出现错误，结果为逻辑 0。这里的输出 P 对应图 4.25 中的状态机输入 p。

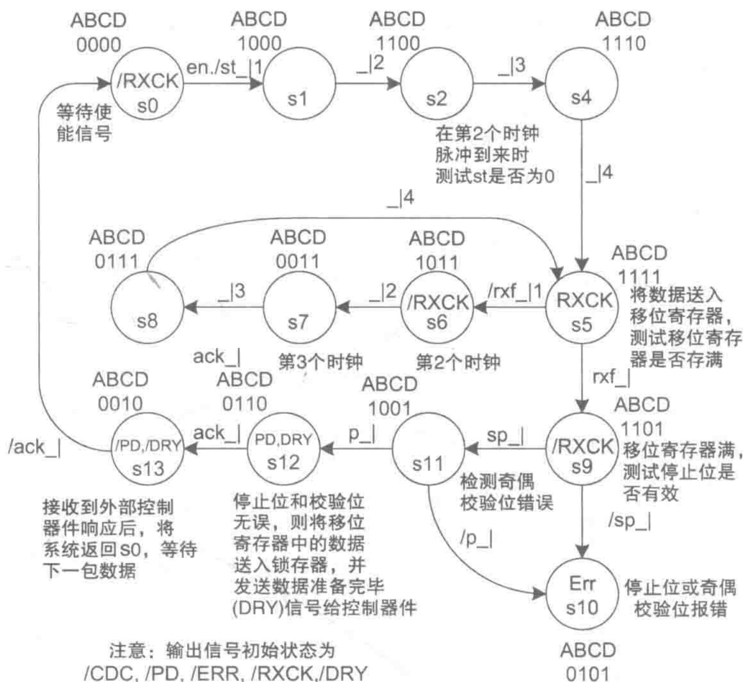


图 4.26 带有奇校验的状态图

图 4.25 中，系统在状态 s9 通过检测停止位来确认数据的完整性；在状态 s11，系统读取奇校验的结果。这两组检测其中任何一个出现错误，系统都将进入状态 s10，等待外围电路发出复位信号。这里还可以根据图 4.21 的原理把起始位的检测也加进来，用与门输出一个最终的检测结果。

4.8.2 图 4.26 对应的 D 触发器公式

下列公式中，变量 P 是奇偶校验的输出（$OP_n = OP_{n+1}$），并作为 FSM 的输入信号 p，参照图 4.23。

A · d = s0 · en · /st + s1 + s2 + s4 + s5 + s8 + s9 · sp

  = /A/B/C/D · en · /st + A/B/C/D + AB/C/D + ABC/D + ABCD + /ABCD + AB/CD · sp

B · d = s1 + s2 + s4 + s5 · rxf + s7 + s8 + s9 · /sp + s10 + s11 + s12 · /ack

  = A/B/C/D + AB/C/D + ABC/D + ABCD · rxf + /A/BCD + /ABCD + AB/CD · /sp + /AB/CD + A/B/CD + /ABC/D · /ack

C · d = s2 + s4 + s5 · /rxf + s6 + s7 + s8 + s11 · p + s12 + s13 · ack

= AB/C/D + ABC/D + ABCD · /rxf + A/BCD + /A/BCD + /ABCD + A/B/CD · p + /ABC/D + /A/BC/D · ack

D · d = s4 + s5 + s6 + s7 + s8 + s9 + s10 + s11 · /p

= ABC/D + ABCD + A/BCD + /A/BCD + /ABCD + AB/CD + /AB/CD + A/B/CD · /p

输出的公式和图 4.22 所给出的一样，除了

ERR = s10 = /AB/CD

PD = dry = s12 = /ABC/D

RXCK = s5 = ABCD

FSM 的仿真结果如图 4.27 所示，图中状态机的测试顺序为：s0、s1、s2、s4、s5、s6、s7、s8、s5、s9、s11、s12、s13、s0、s1、s2、s4、s5、s9、s10、s0、s1、s2、s4、s5、s9、s11、s10。这样状态机所经过的所有路径都参与了测试。

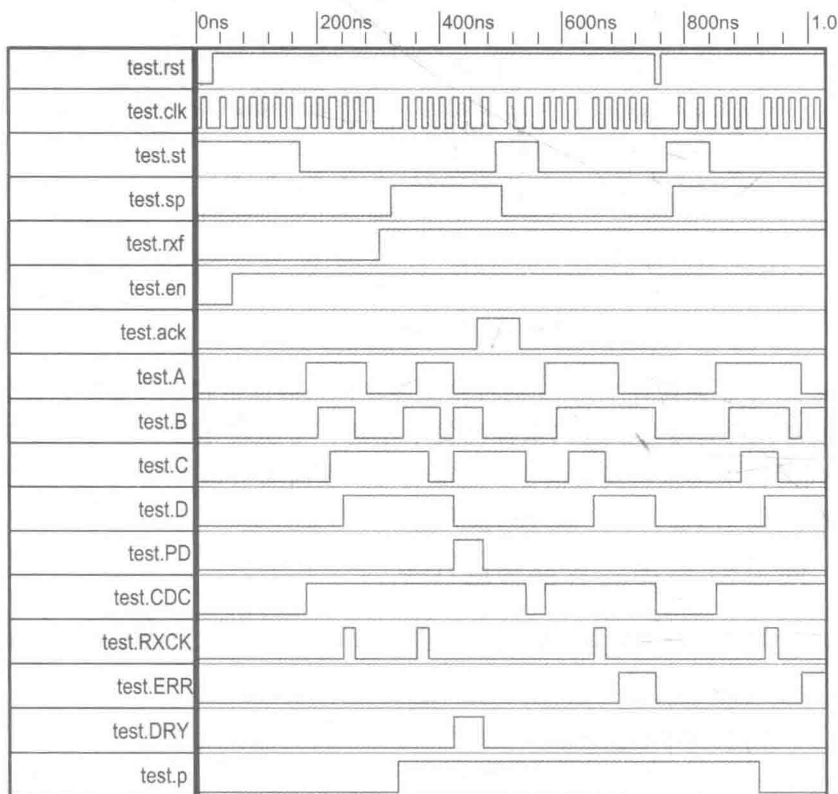


图 4.27　FSM 仿真结果

在进行系统的整体测试之前，系统内的其他模块也需要经过测试，即移位寄存器模块、4 位计数器模块、奇偶校验模块等。

4.9　异步串行发送系统

完成了异步串行接收系统，再来看异步串行发送系统，两者结合起来就是完整的异步串行通信系统了。图 4.28 为异步串行发送系统的框图。

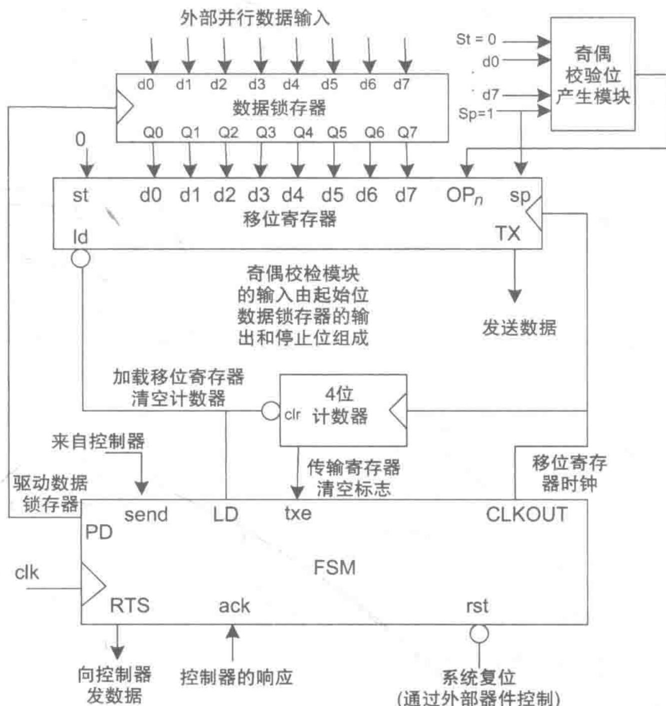


图 4.28　异步串行发送系统框图

锁存器将要发送的数据放在其总线输出端，FSM 通过信号 LD 将起始位和停止位预设为固定的值。这里有必要提醒一下，发送数据的时钟没有必要刻意降低频率，原因在于发送数据时，移位寄存器的时钟可以由 FSM 直接提供。

发送数据的整个过程中，起始状态是外围电路将被发送数据送到并行端口。FSM 将数据加载到移位寄存器中，然后开始逐位向外发送。4 位计数器在一包数据发送完毕后会将信号 txe 拉高。之后 FSM 会发出一个请求发送数据（RTS）信号给外围电路，通知它现有数据已经发送完毕。外围电路随后会通过响应信号 ack 的置高来回应状态机的操作。

图 4.29 给出一套状态机的设计方案。

发送端的移位寄存器所使用的时钟频率必须和接收端的时钟频率一致，这点很重要。如果两个时钟频率不一样，接收端将无法接收数据。即使两个时钟在频率上

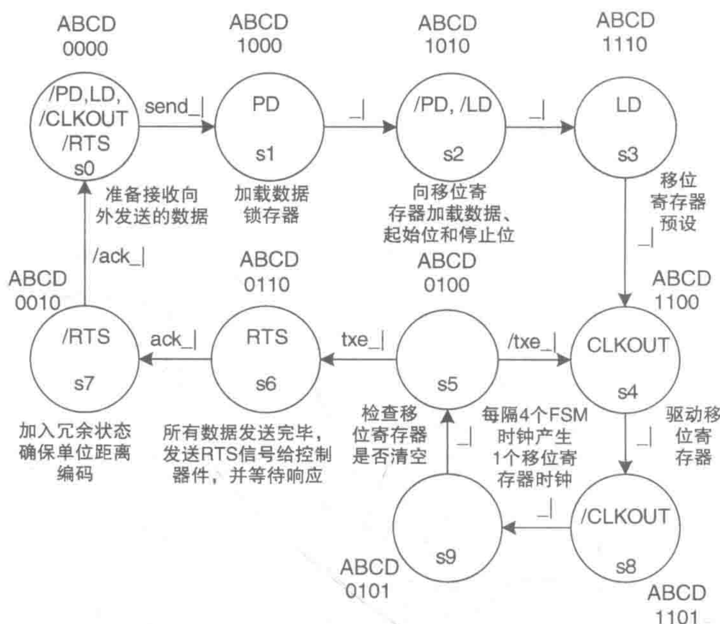


图 4.29 异步串行发送系统状态图

相差很小，也极有可能在验证起始位和停止位等部分发生错误。当发送和接收时钟频率发生偏差时，会影响到整包数据的传输，因此造成 1 位或多位数据的丢失。事实上，要做的就是确保起始位和停止位被正确地传输和接收。

举例来说明，如果发送的时钟频率为 1MHz（通常也用波特率来表示），可以得出整包数据传输所需时间为 $11 \times 1/(1 \times 10^6) = 11 \times 1\mu s = 11\mu s$。在接收端所用的时钟频允许存在一定的偏差，这是因为数据是在 4 个时钟周期的窗口里采样的（见图 4.20），因此两包数据之间是允许一些误差存在的。

在市场上卖的一些通用异步收发（Universal Asynchronous Receiver Transmitter, UART）器件里，一般采样频率窗口所涵盖的系统时钟周期数达到 16 个，而不是我们上面讲的 4 个，这样接收端移位寄存器所用的时钟（RXCK）可以拥有更高的数据采样精度。

通常来说，如果发送端和接收端的时钟精度都很高（类似晶振的那种频率精度），正常情况下是没问题的。在接收端要使得移位寄存器拥有更高精度的时钟，需要修改图 4.22 和图 4.26 中的状态构架，状态循环 s5 ~ s8 中需要加入更多的状态，同时起始部分 s1 ~ s5 也需要加入更多的状态。然而这样的方法是运用独热编码技术来实现的，具体会在第 5 章详细讲解。

因此本例中只需要记住 FSM 时钟频率是波特率的 4 倍。

异步串行发送器的状态图如图 4. 29 所示。这里，在状态循环 s4、s8、s9 和 s5 之间，移位寄存器时钟频率是 FSM 的 4 倍。所以对应 1μs 的波特率，FSM 的时钟频率必须是 4MHz。

4. 9. 1　异步串行发送系统公式

A · d = s0 · send + s1 + s2 + s3 + s4 + s5 · /txe

　　= /B/C/D · send + A/B/D + AC/D + AB/D + B/C/D · /txe

B · d = s2 + s3 + s4 + s5 + s8 + s9 + s6 · ack

　　= AC/D + B/C + /AB/D · /ack

C · d = s1 + s2 + s5 · txe + s6 + s7 · ack

　　= A/B/D + /AB/D · txe + /AC/D

D · d = s4 + s8

　　= AB/C/D + AB/CD

　　= AB/C

PD = s1 = A/B/C/D

CLKOUT = s4 = AB/C/D

LD = /s2 = /(A/BC/D)

RTS = s6 = /ABC/D

FSM 仿真结果为图 4. 30。测试顺序为 s0，s1，s2，s3，s4，s8，s9，s5，s4，s8，s9，s5，s6，s7，s0。

如果用 FPGA 芯片运行这里介绍的发送和接收模块，可以做到很高的波特率，如下表所示

FSM 时钟	接收端 RXCK	发送端 CLKOUT	波特率
4MHz	1MHz	1MHz	1 兆波特
8MHz	2MHz	2MHz	2 兆波特
16MHz	4MHz	4MHz	4 兆波特
32MHz	8MHz	8MHz	8 兆波特
80MHz	20MHz	20MHz	20 兆波特

注意发送模块和接收模块使用的是各自时钟电路针对同样的 FSM 产生的时钟频率。

对于高波特率推荐使用 1m 左右的短距离双绞线来传输数据。但是传输线上的衰减、阻抗匹配和干扰等问题也需要考虑到。对此本书将不做详细介绍。

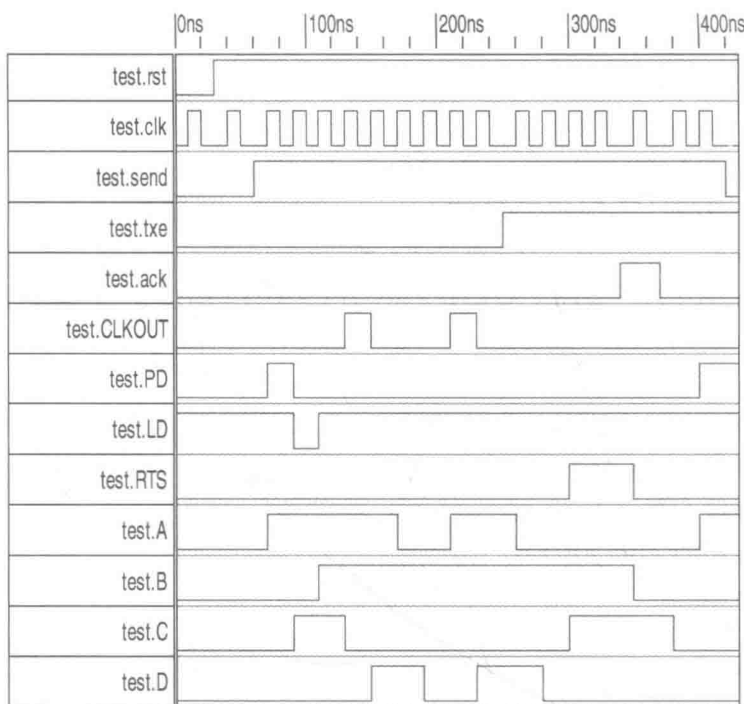


图 4.30 异步串行发送系统 FSM 仿真

4.10 看门狗电路

大部分微处理器如今都有内置的看门狗计时器电路。看门狗电路可通过固定的地址在一定的时间间隔内去改写其内部计数的值，其原理是内部的计时器（一般来说是一个倒数计数器）会不停地被刷新，让它从一个已知的数值开始重新倒数。在下一次刷新到来之前，计数器会一直倒数。如果微控制器在计数器倒数回 0 之前，没有在某个固定的数值将看门狗重置，那么计数器会在到达 0 时重置，并同时复位微控制器。

因此看门狗电路可以作为一个保护电路，防止微控制器在使用过程中由于临时的电源系统故障而导致程序跑飞（例如突然跳转到某一行，而不是当前正在运行的程序）。

另一个用途是在基于微处理器的系统中，操作系统（有可能是实时操作系统）可以定时复位看门狗，提供一种检验微处理器系统可靠性的方法。

编写的微处理器程序需要在固定的时间点将看门狗重置，以防其将整个系统复位。

尽管大部分微控制器都有这项功能，但很多微处理器系统并不具备。因此，需要自己来设计看门狗电路。

图 4.31 所示的时序图是一个可以实现看门狗基本功能的电路设计方案。方案的设计需要根据微处理器的内存读写循环来完成。图 4.31 中所示的内存写循环占用了 4 个时钟周期，即 T1 到 T4。

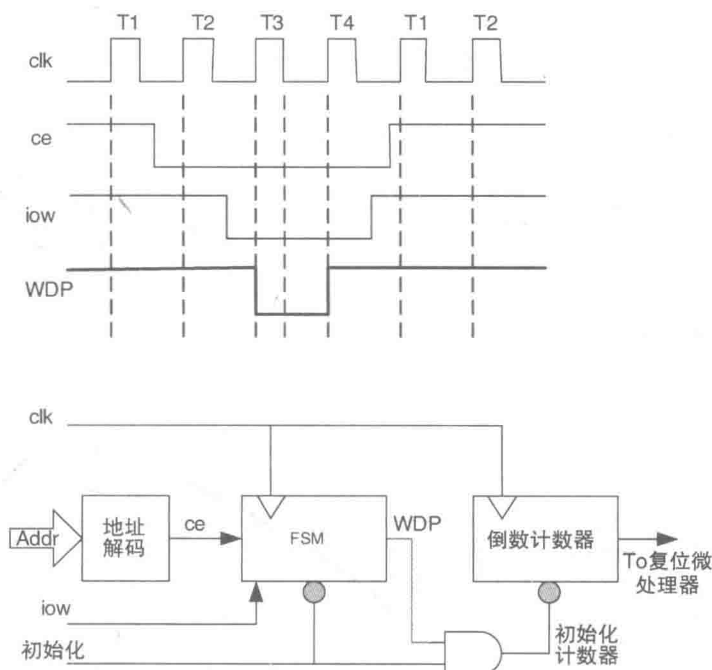


图 4.31　微处理器内嵌看门狗电路系统框图

看门狗电路是由 FSM 来控制的。FSM 通过地址解码电路来监视片选信号 ce 的状态，通常片选信号有效时，会指向微处理器的某一个特定的地址空间。同时微处理器控制的 iow 信号也在 FSM 的监视范围内。当微处理器的地址指向看门狗电路时，ce 信号被拉低，然后在 T2 所对应的时钟周期内 iow 信号也被拉低。在 T3 周期的上升沿，产生看门狗脉冲（WatchDog Pulse，WDP）。这里要求 FSM 必须在固定的时钟周期内（这里的 T3 周期）产生看门狗脉冲。FSM 和看门狗电路所用的倒数计数器都是由微处理器内部时钟 clk 驱动的。

在本书附录 B，B.1 介绍了二进制倒数计数器的设计方法。为了向计数器提供一个固定起始点（从某个数开始倒数），可以用并行加载计数器（参考 B.3 节），将倒数计数器的触发器预先设定一个已知的值。这和图 4.31 中输入信号 initialize（初始化）的目的是一致的（从根本上说就是一个倒数计数器的并行加载输入信号）。

要注意的是，同样的输入信号也让控制看门狗的 FSM 进入初始状态（即状态 s0）。由于看门狗电路需要频繁地向倒数计数器输入初始化脉冲信号，因此计数器不会倒数到 0（否则会引起微处理器重置）。

图 4.32 给出了相应的状态图，FSM 在状态 s0 等待微处理器选择看门狗电路所对应的地址。此后 ce 信号在内存读写循环 T1 周期被拉低（见图 4.31），在 T2 周期的上升沿 FSM 将进入状态 s1。这里 FSM 会等待微处理器将信号 iow 拉低，随后在下一个时钟脉冲（T3）状态机会进入状态 s2，并将看门狗输出信号 WDP 拉低。在下一个时钟周期（T4），FSM 将进入状态 s3，将信号 WDP 拉高，并等待信号 ce 被拉高。在读写循环的末尾 ce 会被拉高，且 FSM 会在下一个 T1 周期的上升沿检测到 ce 信号的变化。

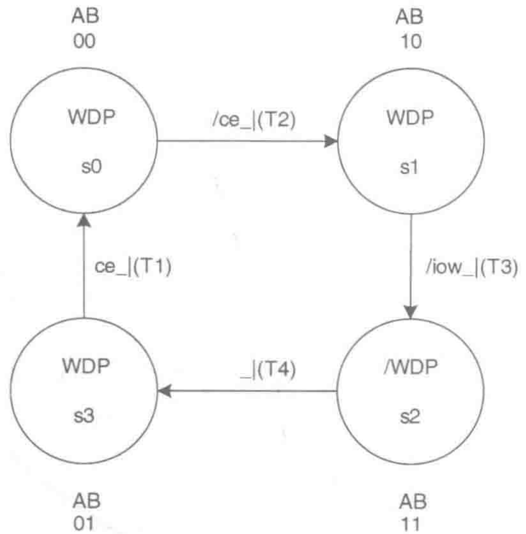


图 4.32 看门狗电路状态图

FSM 的公式是根据图 4.32 来设计的。

4.10.1 D 触发器公式

A · d = s0 · /ce + s1

= /A/B · /ce + A/B

= /B · ce + A/B

B · d = s1 · /iow + s2 + s3 · /ce

= A/B · /iow + AB + /AB · /ce

= A · /iow + AB + B · /ce

4.10.2 输出公式

WDP = /s2 = /(AB)

ce 信号的公式是根据分配给看门狗计时器的地址来决定的。例如，如果分配的地址空间为 300h（二进制为 11 0000 0000），则 ce 的公式可以写为：ce = /(a9 · a8 · /a7 · /a6 · /a5 · /a4 · /a3 · /a2 · /a1 · /a0)

有时候系统会附加识别信号，即在上位机读写端口映射到信号/aen，用来区分是动态存储访问（Dynamic Memory Acless，DMA）循环还是普通的读写循环（参考第 5 章的 DMA 部分）。还有之前提到的 iow 信号被用来识别是否为写循环。

因此 ce 的公式可以重新写为：ce =/(a9 · a8 · /a7 · /a6 · /a5 · /a4 · /a3 · /a2 · /a1 · /a0 · /aen · /iow)

这里将附录 B 里对应的倒数计数器的公式摘抄如下：$Qn \cdot t = \prod_{p=1}^{p=n} /q\ (n-p)$

作为一个 n 阶计数器，第一个 T 触发器的输入 q0 · t 的值为 1。

将此公式展开得到一个四阶倒数计数器：

Q0 · t = 1

Q1 · t = /q0

Q2 · t = /q0 · /q1

Q3 · t = /q0 · /q1 · /q2

Q4 · t = /q0 · /q1 · /q2 · /q3

注意到计数器需要一个异步初始化信号和每一个 T 触发器相连，用于构成并行加载输入逻辑电路（参考公式 B. 4 和图 B. 4）。

图 4. 33 是仿真结果。系统在检测到信号 ce 和 iow 分别依次被拉低后，输出信号 WDP 在状态 s2 被拉低。触发器 A 和 B 很清晰地表明了 FSM 的测试次序。

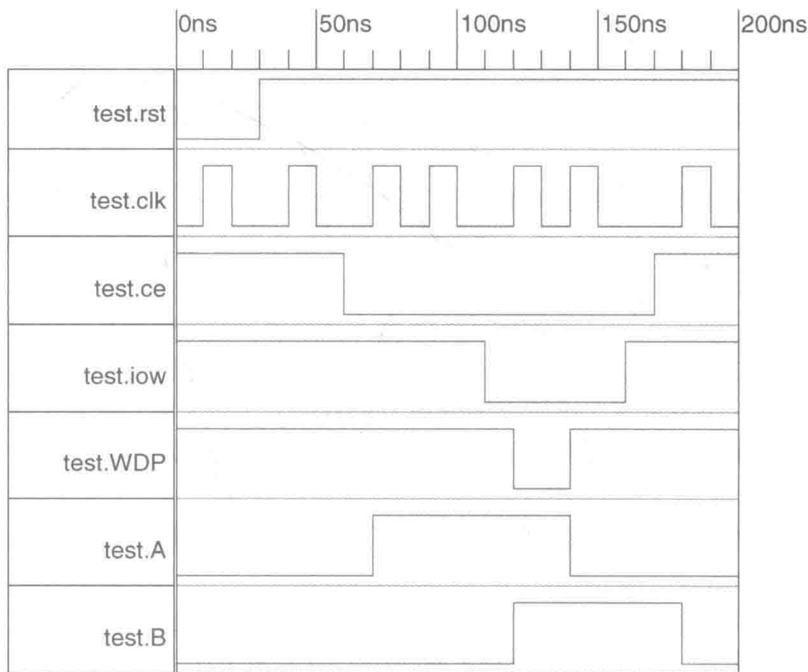


图 4. 33　看门狗计时器 FSM 仿真

值得注意的是图里一开始多出来的时钟脉冲是由测试平台产生的，用于测试 FSM 能够保持在状态 s0 和 s1 直到 ce 和 iow 两个信号的状态满足条件为止。在现实应用中这样的情况是不会发生的，因为微处理器完全掌控信号 ce 和 iow 的状态。

4.11 小结

本章通过使用系统框图、状态图以及前 3 章介绍的方法向大家介绍几个实际案例，它们都是基于 D 触发器来设计的。读者可以根据需要将介绍的案例运用到各自的系统设计当中，或者将它们进行扩展，以满足更多的个性化需求。

在下一章，将向大家介绍用每一个 D 触发器对应一个状态的设计方法，这种方法的特点是将不再需要二次状态变量。

第5章　运用独热编码技术设计FSM

5.1　独热编码简介

在这之前介绍的 FSM 都是运用二次状态变量，参与设计每一个状态。这需要使用单位距离编码的格式，目的是让系统在输出端保持稳定。

另一种方法就是将每一个状态都用一个触发器来表示。尽管这可能会被认为是一种资源的浪费，但从理论上来说，它能够避免输出端的一些潜在的问题，因为每一个状态都有其独立的触发器。这样在任何一个时刻，只有一个触发器被激活，即 FSM 此刻所对应的状态。

这种设计理念被称之为“独热”（One - Hotting）编码，并且在基于 FPGA 芯片的设计中被广泛采用。这是因为 FPGA 的内部结构可以通过编程来获得很多触发器单元或者门电路。因此产生大量的触发器不是一件难事。而对于一个可编程逻辑器件（PLD），它的内部结构只允许其拥有一定数量的触发器，并且受到由与门/或门等逻辑门所组成的“乘积项之和”的控制。

独热编码技术的另一个特性就是它不需要复杂和多层次的逻辑电路，因为不需要定义新的状态变量，除了主要的输入信号和前一个（或几个）状态，这样逻辑电路的速度会相对较快。

下面将为大家介绍用独热编码来实现 FSM 的方法。

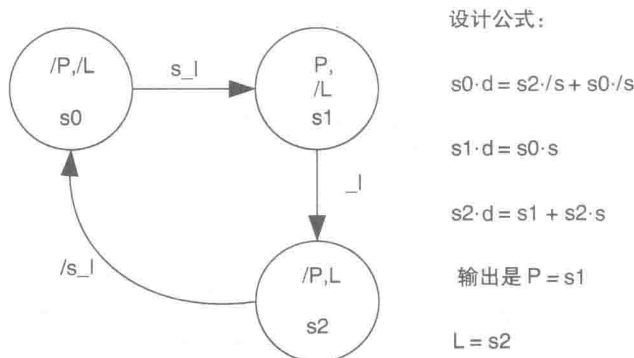
请看图 5.1，在使用独热编码的情况下，再次看一下之前讨论过的带有指示功能的单脉冲发生器。它使用了 3 个状态（原来的版本用了 4 个状态），这是可以实现的，因为它不需要使用单位距离编码。因此，二次状态变量也就没有必要考虑了。

图 5.1 右边是 FSM 的公式。大家可以从独热状态图中推算出每个表达式的由来。

起先，FSM 应该在状态 s0。这可以通过设置一个起始的输入信号，这样代表状态 s0（FFS0）的触发器就被激活了，所有其他的触发器（FFS1 和 FFS2）就被重置。

先来分析状态 s0。此刻 FSM 应该仍然处于状态 s0，当输入信号 s 被置 1 之后，离开状态 s0 的条件便得到满足。

然而，在 FSM 离开状态 s0 之前，触发器 FFS0 的输入端 D 需要一个信号，让这个触发器保持激活状态。这个信号可以表示为：s0 · /s。



状态图无需任何二次状态变量，因为每一个状态都表示一个D触发器

在初始化过程中，代表状态s1和s2的触发器被复位，代表s0的触发器被激活

图 5.1　使用独热编码的一个例子

这是因为从状态 s0 的“离开条件”是输入信号 s 从逻辑 0 变为逻辑 1，所以当输入信号 s 没有被置 1，即 s = 0，或者/s 时，状态 s0 对应的触发器必须处于激活状态，即系统停留在状态 s0。

因此，s0 · /s 可以被认为是“状态保持表达式”，因为它起到的作用就是让触发器 FFS0 被激活，直到需要进入下一个状态 s1。

同时，当 FSM 到达状态 s2 时，只有当输入信号 s 的值被拉低（从逻辑 1 变为逻辑 0）时，才会从 s2 回到 s0。因此，这里还有另一个表达式：s2 · /s。

这个表达式就是所谓的“触发表达式”，或者可以称之为“激活”表达式。

因此状态 s0 的触发器 FFS0 的完整表达式为如下所示：s0 · d = s2 · /s + s0 · /s。

现在分析状态 s1，FSM 进入 s1 的条件是处于状态 s0 且 s = 1，所以触发器 FFS1 的表达式则为：s1 · d = s0 · s。

而离开状态 s1 的条件就是一个简单的时钟脉冲。从状态 s1 进入到 s2 是没有输入信号作为触发条件的；所以，当 FSM 进入状态 s1，它将在下一个时钟脉冲（上升沿）到来时自动进入状态 s2，而不需要所谓的“状态保持”。

再看最后一个状态 s2。进入状态 s2 的条件就是状态 s1，因为没有输入信号的变化作为触发条件。而在状态 s2 和 s0 之间却存在着“状态保持”表达式，即：s2 · s。

当 s = 1 时 FSM 必须停留在状态 s2。所以触发器 FFS2 的表达式为：s2 · d = s1 + s2 · s。

此外，输出信号可以表示为 P = s1。

因为只有在状态 s1 输出端才被置高；而 L 只有在状态 s2 才置高，因此 L = s2。

整个 FSM 的电路图如图 5.2 所示。注意到系统已经带有初始化逻辑电路。在独热系统中，代表最初状态的触发器必须被激活，而其他所有的触发器必须被重置。如果触发器不带预设电路和复位输入信号，那么必须用同步复位电路来完成初始化工作（例如第 3 章讲稿 3.16 和 3.19 中提供的方法）。

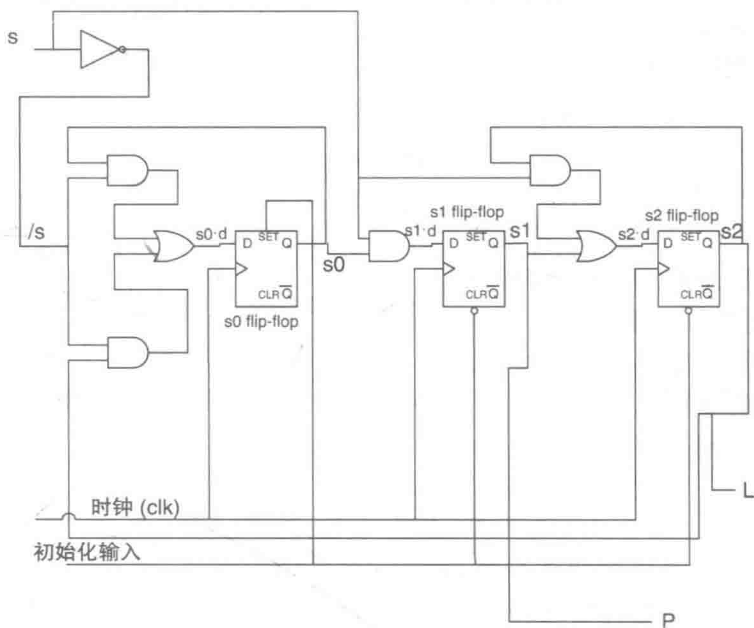


图 5.2　使用独热编码的单脉冲 FSM 电路图

现在请大家看图 5.3。这里状态 s0 的表达式和刚刚介绍的上面的例子是一样的。而对于触发器 FFS1，进入状态 s1 的通道有两个，一个是通过 s0（s0 · st)，还有一个是通过 s3。离开状态 s1 也有两个分支，一个是到 s2（s1 · x)，一个是到 s3（s1 · /x)，将两者结合，可以得到：s1 · d = s0 · st + s3 + s1 · x · /x

简化后得到：s1 · d = s0 · st + s3，因为 s1 · x · /x 实际计算结果为 0。

FSM 会停在状态 s1，由于通往 s2 和 s3 的条件正好相反（互为补码)，即 x 和/x，这正好暗示着 FSM 会停留在状态 s1。当然，这也导致 s1（/x · x）的结果是 0（s1（1 · 0）或者 s1（0 · 1）都为 0)。

观察图 5.3 中的状态图，可以发现一旦 FSM 到达 s1，它将会在下一个时钟周期要么进入 s2 要么进入 s3，而没有理由留在状态 s1。所以再次验证了第二个表达式（s1 · d = s0 · st + s3）是正确的。

注意：在遇到有互补输入信号形成分支的 FSM 里（在图 5.3 里即 x 和/x)，推导出来的表达式一般会被抵消。而图 5.3 中其他状态的表达式则为一般常见的情况。

现在看图 5.4 中给出的 FSM 的状态图。里面同样有两条分支，但是这次分支

的触发条件不再是互补的。注意到表达式 s1 · d 含有一项：s1 · (/x · /y)，这一项将 FSM 保持在 s1，直到 x 或者 y 其中任意一个的值变为逻辑 1，即如果 x 和 y 的值都为 0，FSM 会永远停留在状态 s1。

注意：当使用不同的输入来触发两个分支时（这里的 x 和 y），两个输入必须相互独立。

图 5.4 中，独立的状态 s3 是通过 s1 进入的，但是一旦进入到 s3，就无法出去了。FSM 会停留在 s3 直到被初始化，重新回到状态 s0。因此，写状态 s3 的表达式时，等号右边也需要加入 s3。

大家试着将图 5.5 的状态表达式写出来，不要忙着看下面的答案。

独热编码技术适用的场合是使用 FPGA 芯片来设计大型状态机，因为 FPGA 芯片内部可以配置大量的触发器，而且门电路级的状态机表达式也相对简单。

本章的剩余部分将介绍一些较为复杂的 FSM，并使用独热编码的技术来设计它们。这些例子主要是将微控制器融合到系统中，并使用 FSM 来控制整个系统。每个例子都会对读者今后独立设计数字系统有所帮助。

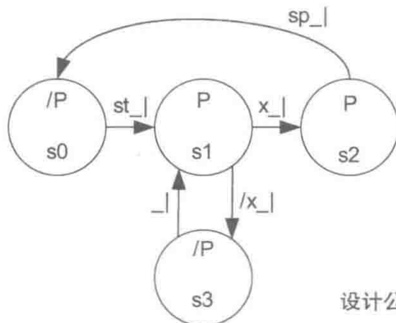


图 5.3 带有两路分支的状态图

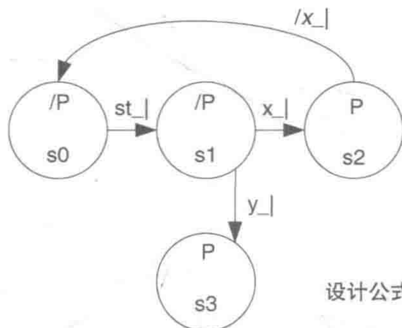
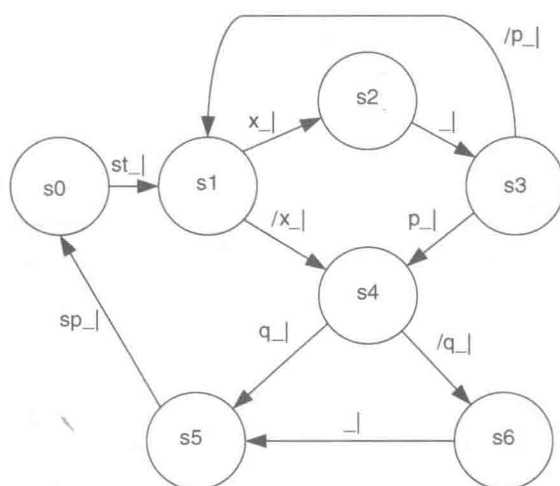


图 5.4 输入条件不互补的两路分支状态机

5.2 数据采集系统

微控制器或者数字信号处理芯片已经被广泛运用到数据采集系统（Data Acquisition System，DAS）中。在使用微控制器的系统中，模 - 数转换单元（ADC）会被集成到微控制器芯片中去。这时候，从 ADC 里面转换出来结果会被系统取整后



答案:

s0 · d = s5 · sp + s0/st

s1 · d = s0 · st + s3 · /p

s2 · d = s1 · x

s3 · d = s2

s4 · d = s1 · /x + s3 · p

s5 · d = s4 · q + s6 + s5 · /sp

s6 · d = s4 · /q

图 5.5　留给读者的思考题

使用。如果数据采集系统需要较高的计算速率，那么就要使用到数字信号处理芯片了。此时，既可以使用整数计算电路处理的结果，也可以选择浮点运算处理器来获得“真实”数据。

数据采集系统的一大特性在于其处理数据的速度是有限的，而且和它使用的处理器的处理速度有很大关系。从某种程度上来说，这种限制可以使用并行数据处理和硬件算法电路来改善。

围绕着 FSM 的硬件设计可以是地址计数器、减法器、乘法器以及除法器等，FSM 可以对它们形成完全的控制。这样设计增大了系统的数据输出量。如果不需要系统进行“实时”的数据处理，可以用 FSM 来搜集并存储数据，这样微控制器或者数字信号处理芯片可以在以后的时间里处理这些数据。

在第 2 章曾经看到过数据采集系统的例子，这里用图 5.6 再复习一下。系统的基本功能是使用并行模 - 数转换器（flash ADC），以获得很高的数据转换速率。而

且系统带有高速静态随机存储芯片（Static RAM，SRAM），用来存储被转换后的数据。这样的设计目的是为了和外部系统形成互动，例如外部系统向 FSM 发送启动输入信号 st，FSM 向外部系统发送存储芯片空间已满的信号（f）等。

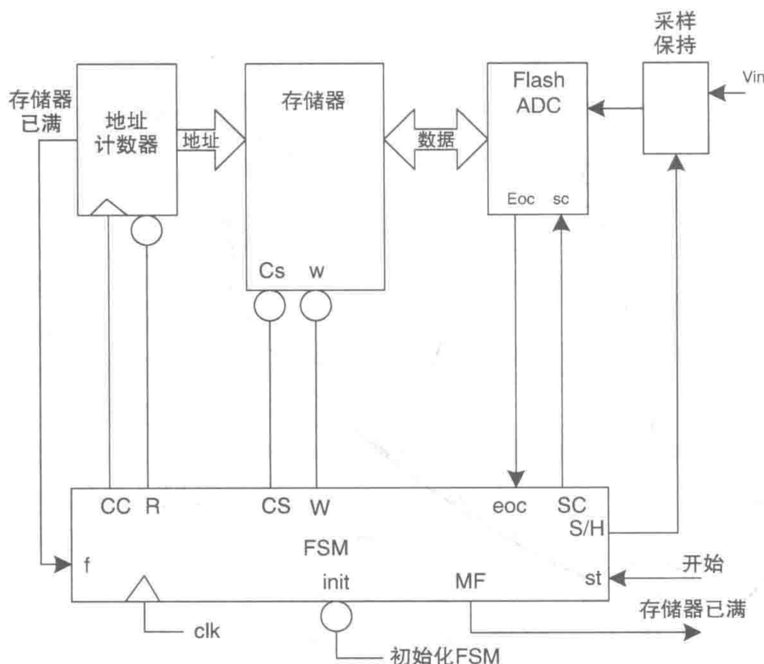


图 5.6 高速数据采集系统

现在，可以绘制数据采集系统的状态图了，如图 5.7 所示，大家会发现图 5.7 和第 2 章讲稿 2.4 ~ 2.10 所介绍的状态图差别不是太大。图 5.7 的系统流程很清晰，当外部输入信号 st 被激活，系统会开始工作，并执行下列命令：

1）采样和保持电路被设置为保持状态，并准备接收模 - 数转换器的数据（状态 s1）。

2）模 - 数转换器开始进行数据转换，FSM 会等待一个转换结束标志信号（eoc）被拉高，表示转换器完成了一次转换（状态 s2）。

3）在状态 s3，FSM 会通过拉低片选信号（CS）选择相应的存储芯片。一旦信号 eoc 回到逻辑 0，FSM 会立刻进入状态 s4。

4）在状态 s4，FSM 会将存储芯片的写信号 W 激活（拉低）。

5）在状态 s5，存储芯片的写信号被拉高，数据被写进存储芯片。

6）在状态 s6，片选信号被拉高，存储芯片被释放。这一步的要点是确保存储芯片在地址计数器变化（选择下一个地址空间）之前被释放。

7）在状态 s7，地址计数器在信号 CC 的驱动下计数。当 CC = 1 时，即在信号

的上升沿，计数器开始工作。并且在 s7 中，系统会检查最后一个地址空间有没有被访问（激活信号 f 表示所有空间都被访问），如果存储芯片还没有满，系统会从 s1 到 s7 再开始循环。

8）当存储空间被存满以后，FSM 会进入状态 s8，并激活 MF 信号通知外部系统。

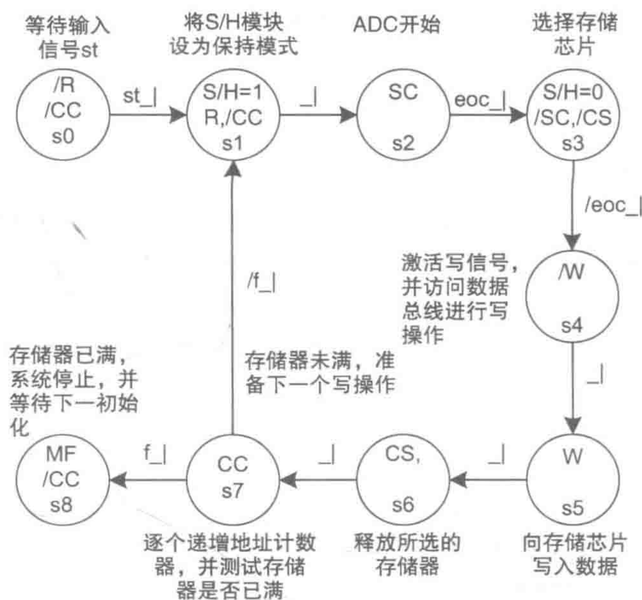


图 5.7　数据采集系统的状态图

注意到 MF 信号可以和一个便携终端发出的中断信号相连，这样系统从 st = 1 时开始工作，在任务完成后被中断。

独热编码的状态公式如下：

s0 · d = /st 触发器 s0 将在系统初始化时被激活，并等待 st 信号被拉高。

s1 · d = s0 · st + s7 · /f

s2 · d = s1 + s2 · /eoc

s3 · d = s2 · eoc + s3 · eoc

s4 · d = s3

s5 · d = s4

s6 · d = s5

s7 · d = s6

s8 · d = s7 · f + s8 系统进入 s8 后将在状态 s8 保持住，直到系统被重置。

系统输出的表达式为：

S/H = s1 + s2

SC = s2

CS = /(s3 + s4 + s5) 这个信号是低有效，从状态 s3 到 s5 输出；

W = /s4 这个也是低有效信号，只在状态 s4 输出；

MF = s8

R = /s0 低有效信号，只在状态 s0 输出；

CC = s7 当系统进入状态 s7 之后，将会把 CC 信号拉高，在离开状态 s7 的之后，此信号被拉低。

这些输入输出信号可以用来产生 Verilog 代码文件并进行仿真，如图 5.8 所示。从图 5.8 可以看出，系统循环运行了 4 次，在第三次时进入到了状态 s8 并停止。注意观察控制存储芯片的片选信号和写信号，以及地址计数器的脉冲。还有，在最后一个循环，存储芯片被写满的标志信号 MF 在状态 s8 被拉高。随后复位信号将 FSM 返复位到状态 s0。

图 5.6 中的系统只能向存储芯片写数据，并不具备读的功能。读者可以将此系统加以修改，让系统存储的数据可以被读出来，但要实现这样的功能，还必须考虑到其他一些情况。

下一节向大家介绍如何对存储芯片进行读写。

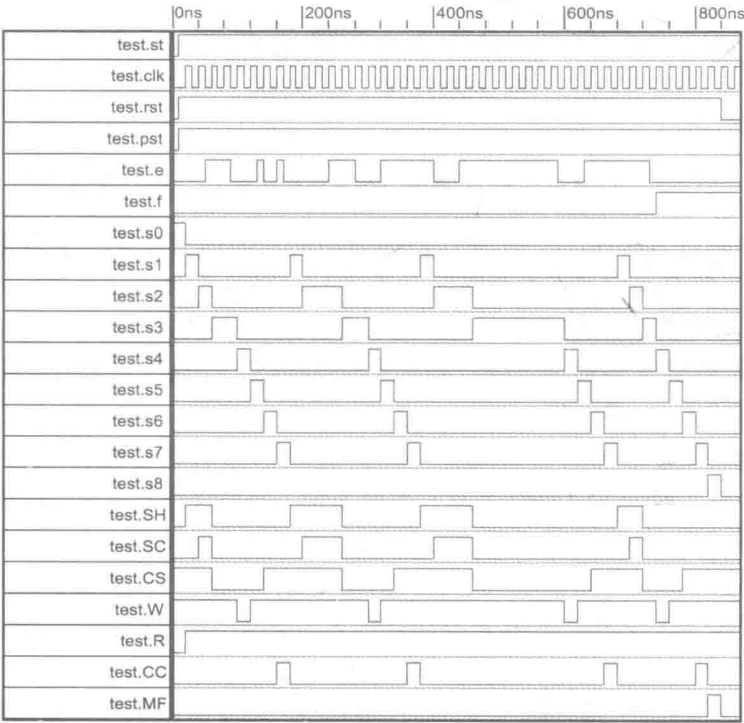


图 5.8　数据采集系统 FSM 控制器仿真波形

5.3　内存共享系统

除了主控单元外，存储芯片通常也需要被其他的外部系统访问。例如，有可能是一个外部的单片机，需要处理存储在内存里的数据。图 5.9 所示的系统构架诠释了这种情形。在这样的系统里，访问存储芯片的可以是 FSM，也可以是其他的外部系统（例如单片机或者数字信号处理系统）。此时，存储芯片变成了“共享”单元。系统运行的理念是：当需要往内存里存储数据时，只有 FSM 可以访问存储芯片，并将数 - 模转换器的结果存入内存。其他情况下，外部系统可以访问存储芯片，但条件是里面存入了可读的数据。

RMA（Read Memory Available，内存可读）信号一旦被拉高，则表明 FSM 和存储芯片已经断开，外部系统随后才能访问存储芯片。当外部系统完成了对存储芯片的读操作之后，和存储芯片断开的同时，需要向 FSM 发送一个响应信号 ACK，这样 FSM 在收到这个信号之后，可以恢复到它的初始状态。系统中 FSM 是主控单元，RMA 信号和 ACK 信号组成了一组通信握手机制（协议）。

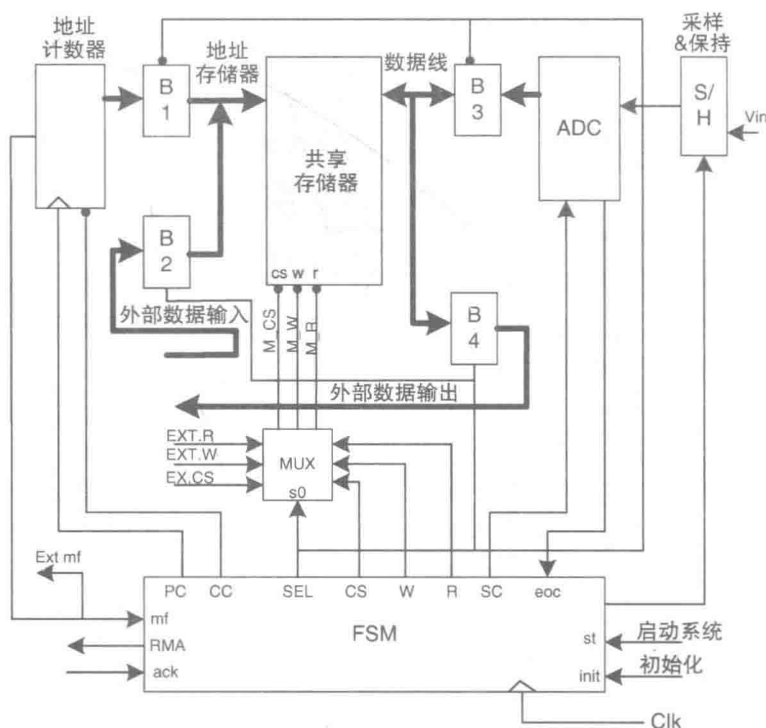


图 5.9　内存共享系统框图

需要注意的是，FSM 会使用 SEL 信号去控制三态门缓冲器 B1、B2、B3 和 B4。在图中可以发现当 SEL 信号为 0 时，B1 和 B3 被选中（有效），SEL 为 1 时 B2 和 B4 被选中。不同的缓冲器连接了不同的外部系统。

正是这些三态缓冲器和存储芯片之间的连接，让存储芯片成为可以被“共享”的资源。

1）三态缓冲器 B1、B2、B3 和 B4 分别和数据、地址总线相连。双向复位选择器 M 用来选择存储芯片的控制来源（FSM 或者外部系统的控制信号）。

2）当 M 的控制输入信号 s0 = 0 时，FSM 的 3 个信号 CS、W 和 R 与存储芯片连通。当 s0 = 1 时，外部系统将通过这 3 个信号来控制存储芯片。

复位选择器 M 的行为可以用以下公式来描述：

M _ CS = CS · /SEL + EXT _ CS · SEL

M _ W = W · /SEL + EXT _ W · SEL

M _ R = R · /SEL + EXT _ R · SEL

要注意握手协议 RMA 信号和 ACK 信号在本系统里是不可或缺的，因为如果外部系统没有从 FSM 接收到 RMA = 1 的信号，是不能获得访问存储芯片的权限的。同样，外部系统必须首先断开和存储芯片的连接，才能将 ACK = 1 发送给 FSM。

系统的状态图（见图 5.10）和图 5.7 十分相似，但加入了控制存储芯片的信号。

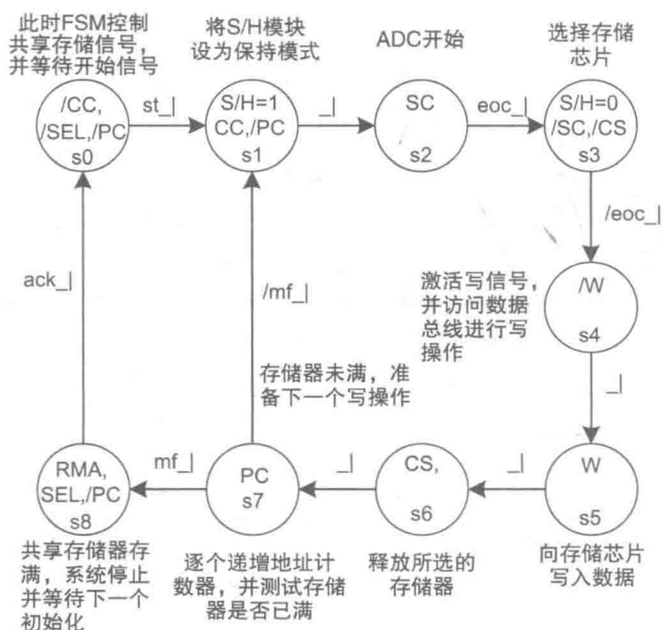


图 5.10 内存共享系统的状态图

需要指出的是，在图 5.10 中，我们假设了模 - 数转换器（ADC）的处理速度比 FSM 从状态 s3 经过一个循环回到状态 s2 慢，并且 FSM 要在状态 s3 等待信号 eoc 回到逻辑 0 才能进入状态 s4。

下面给出了表示状态图的公式。

信号 d 为 D 触发器的输入：

s0 · d = s8 · ack + s0 · /st

s1 · d = s0 · st + s7 · /mf

s2 · d = s1 + s2 · /eoc

s3 · d = s2 · eoc + s3 · eoc

s4 · d = s3 · /eoc

s5 · d = s4

s6 · d = s5

s7 · d = s6

s8 · d = s7 · mf + s8 · /ack

输出公式：

CC = /s0　　低有效输出

SEL = s8

RMA = s8

S/H = s1 + s2

SC = s2

CS = /(s3 + s4 + s5)　　低有效输出

W = /s4　　低有效输出

PC = s7　　假设地址计数器是用上升沿触发的。当 FSM 离开 s7 以后，PC 信号恢复到其释放状态（PC = 0）。

5.4　简易波形发生器

本节中的案例将涵盖一部分系统设计里常见的问题，包括建立微处理器或者微控制器和 FSM 之间的连接所涉及的各个方面的注意点。

这里将讨论用 FSM 来设计一个频率合成器（波形发生器）。主要想法是通过一个并行数据端口，将一组数据从微处理器或者微控制器传送到一个存储芯片里，之后 FSM 从对应的存储芯片地址，将数据读出并将其送入数 - 模转换器（DAC）。系统框图如图 5.11 所示。

要注意的是波形数据可以是任意的波形采样数据，其形状取决于它们的周期和采样频率。因此，用于指示存储芯片已经写满的 mf 信号，实际上也就是“波形结束”的信号，它是通过比较地址总线的当前值和控制器件发出的“地址总线上限”

值是否一致而产生的。

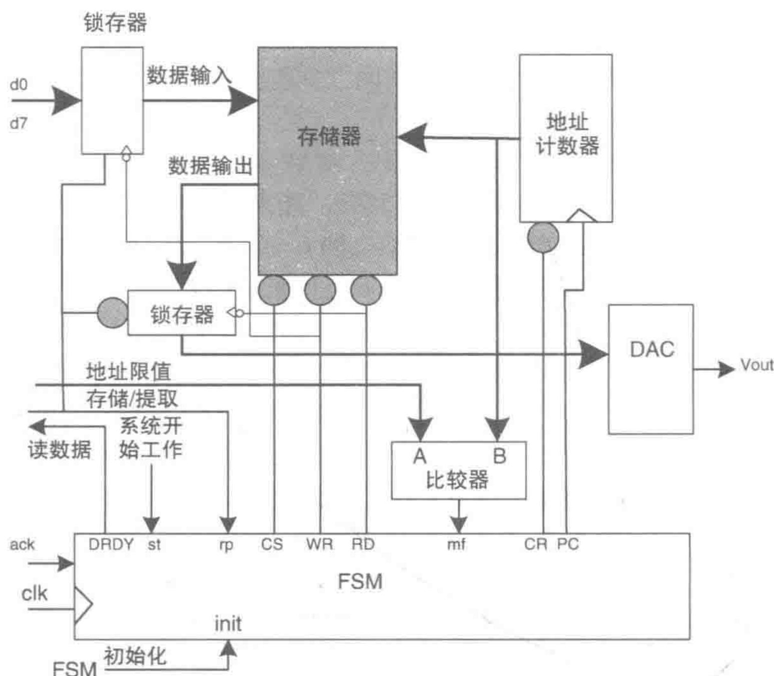


图 5.11 波形发生器系统框图

这里需要指出的是，波形的采样数量必须和存储芯片的大小匹配，采样数据只要小于等于存储芯片的空间即可。但是必须规定好采样的结束点，这样 FSM 读取存储芯片时，从起始地址空间开始循环读取，并不断地将数据送到数 - 模转换器 (DAC)，输出端看到的波形才是连续的。

图 5.11 里所用到的并行数据端口连接微控制器和存储芯片，并将采样数据送到存储芯片里。输入信号 st 表示系统开始工作，输入信号 rp 是用来定义系统的工作状态，当 rp = 1 时为数据存储模式，rp = 0 时为数据提取模式。这两个输入信号可以由微控制器产生，也可以直接使用按钮来触发。

5.4.1 工作原理

系统加载电源之后，FSM 等待信号 st 被激活。当信号 rp 被置 1 后，FSM 将输出信号 DRDY 激活（置 1)，告诉微控制器系统正在等待数据。然后微处理器会将数据放到并行输出端口 d0 ~ d7。FSM 紧接着将这一个字节的数据写到存储芯片中，并将信号 DRDY 拉低，告诉微控制器数据已经处理完毕。微控制器看到信号 DRDY 被拉低后，便会给出一个 ack 响应信号（拉低)，告诉 FSM 一个完整的数据传输过程已经结束。整个过程会不断地重复进行，直到存储芯片被写满。而波形采样的个

数，跟存储芯片什么时候被填满密切相关。FSM 根据微处理器给出的地址上限，判断存储芯片是否存满，一旦存满，FSM 的输入信号 mf 将被置 1。

如果输入信号 rp 被设置为数据提取模式（rp =0），FSM 接着会开始将存储芯片里的数据不断地向数 - 模转换器（DAC）发送。信号 st 被释放之前，波形将一直显示在用户终端。

基于上述功能的状态图以及使用“独热”解码设计的状态机，将在下一节重点讨论。

5.4.2 解决方案

这是一个相对复杂的设计案例，通过对微控制器的编程，利用并行数据总线来完成系统控制。

状态图由两个循环组成：一个是存储数据模式，另一个是回放模式。运用米利型 FSM，系统状态图由 13 个状态组成，如图 5.12 所示。

当然，其他类似的解决办法也是可行的，不过可能需要更多的状态来完成整个系统的设计（例如利用摩尔 FSM 的输出形式）。而运用米利 FSM 的好处是 FSM 的主体既可以用来运行读数据的操作，也可以用来运行写操作。读信号 R 和写信号 W 都是低有效，具体的操作模式大家可以参考讲稿 3.26。

现在来简要说明一下整个状态图的运作。

当开始信号 st 被激活后，FSM 离开状态 s0 进入 s1，在 s1 中系统将重置地址计数器。下一个时钟上升沿到来时，系统进入状态 s2，并将 DRDY 信号拉高，表示系统已经准备好。当微控制器收到 DRDY 信号之后，它将三态数据总线的使能位激活，将并行数据总线和存储芯片的数据总线相连，当 rp =1 时，数据可以被写入存储芯片。与此同时，另一个用来读取存储芯片数据的三态总线被禁用。微控制器此时会将 ack 信号拉高，FSM 便会进入状态 s3，存储芯片的片选信号会被激活（CS =0），此时存储芯片被选中，接着当 FSM 进入状态 s4 之后，由于此时信号 rp =1（即数据存储模式），存储芯片的写信号 WR 会被拉低。注意到如果存储芯片是在数据提取模式（即 rp =0），在状态 s4 中被拉低的会是读信号 RD。进入到状态 s5 之后，WR 信号（或者 RD 信号）将被拉高，用以对相应的地址空间进行写操作（或者读操作）。

FSM 会在下一个时钟到来的时刻进入状态 s6，将片选信号 CS 拉高，此时存储芯片将被释放。在下一个状态 s7，PC 信号将被拉高，地址计数器将被触发。此时系统将测试存储芯片是否已经被写满。如果存储芯片没有满，FSM 将从状态 s7 跳到状态 s9，此刻系统如果在数据存储模式（rp =1），信号 DRDY 将被拉低，并等待从微控制器发出的响应信号 ack（此信号标志着微控制器已经将下一个要写入存储芯片的字节准备好）。进入状态 s12 之后，状态机会回到状态 s2，进入下一个地址空间的循环操作。注意，通常情况下，在离开状态 s7 时，PC 信号就会被拉低。

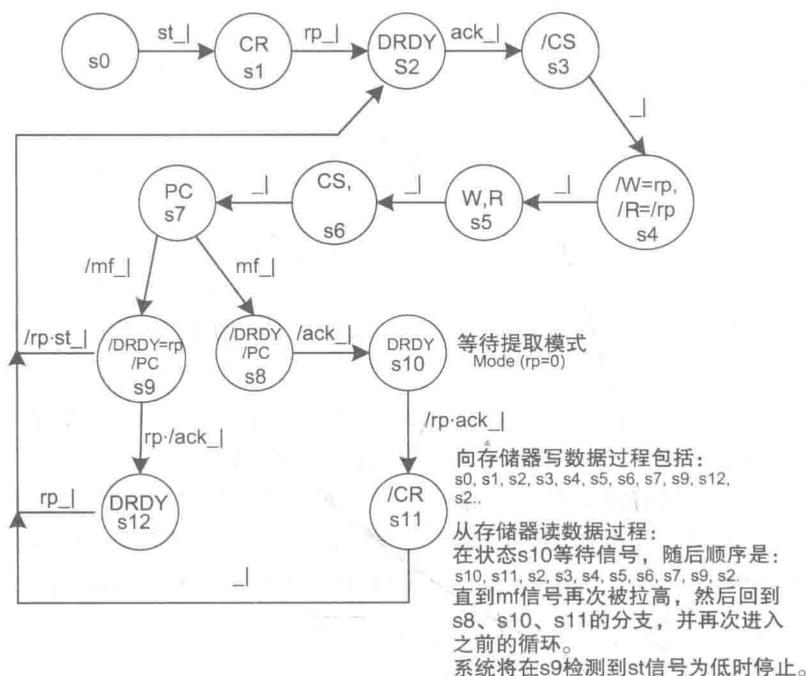


图 5.12 使用“独热”解码设计的波形发生器 FSM

整个过程将持续到存储空间被全部写满。当存储芯片存满之后，FSM 会从 s7 进入 s8，而不是 s9，并将 DRDY 信号置低，等待微处理器的响应。当收到响应信号 ack 之后，FSM 会在状态 s10 等待用户将输入信号 rp 置 0（表示此时系统切换到数据提取模式）。

在数据提取模式，FSM 会进入 s11，将地址计数器重置，接着便回到状态 s2 开始进行 s2，s3，s4，s5，s6，s7，s9，s2 的循环，循环的条件为 rp = 0 和 st = 1。在这样的循环中，存储芯片会被一直读取，而此时 rp = 0，地址计数器会一直不停地从 0 到设定的上限值循环计数，直到输入信号 st = 0。

针对图 5.11，这里对数据的读写作一个补充说明：数据存储模式下（rp = 1）当 WR 信号被拉低时，FSM 会读取 d0 ~ d7 上的数据，而当 WR 被拉高以后，FSM 将锁存器里的数据通过数据输入总线送往存储芯片。在数据提取模式下（rp = 0）当 RD 信号被拉低时，存储芯片里的数据通过数据输出总线被送往锁存器，随后 RD 信号被拉高，锁存器里的数据通过总线送往数 - 模转换器（DAC）。

需要注意的是，FSM 会在状态 s8 和 s9 等待信号 ack 被释放来完成握手通信。

系统可以增加一个复位信号，作用是无论系统运行到什么状态，都可以被强制复位到状态 s0。

现在可以根据状态图推导独热公式了。

5.4.3 D 触发器输入端 d 对应的方程

s0 · d = s0 · /st　　保持项

s1 · d = s0 · st + s1 · /rp

s2 · d = s1 · rp + s11 + s12 · rp + s9 · /rp · st + s2 · /ack

s3 · d = s2 · ack

s4 · d = s3

s5 · d = s4

s6 · d = s5

s7 · d = s6

s8 · d = s7 · mf + s8 · ack

s9 · d = s7 · /mf + s9 · /(rp · /ack) · /(/rp · st)　　注意带有两路分支的保持项

s10 · d = s8 · /ack + s10 · /(/rp · ack)

s11 · d = s10 · /rp · ack

s12 · d = s9 · rp · /ack + s12 · /rp

5.4.4 输出公式

CR = /(s0 + s11)

DRDY = s2 + s3 + s4 + s5 + s6 + s7 + s10 + s11 + s12

或者 DRDY = /(s8 + s9 · rp) 这是将其看作低有效信号

CS = /(s3 + s4 + s5)

W = /(s4 · rp)

R = /(s4 · /rp)

PC = s7

以上公式都可以用 Verilog HDL 直接描述。

5.5 运用微处理器（微控制器）控制 FSM

为了编写程序，需要首先掌握 FSM 和微控制器之间的接口定义。

从图 5.13 可以看出，微控制器需要一个字节宽度（8bit）的接口，作为输出将波形文件送到存储芯片，并需要额外的两个比特（2bit），用作微控制器和 FSM 之间的握手通信协议。而且还需要一个字节宽度作为输出，来发送存储芯片的上限值。微控制器的主要任务是产生编译器所需要的波形数据。本书暂不讨论如何产生波形数据的文件，但波形中某个点对应的数值，也可以用微控制器来计算产生并发

往存储芯片。

代码 5.1 是微控制器源代码的一部分，所用的语言是 C。当然实现方法是多种多样的，C 语言在微控制器编程里十分常见。

```
//---------列出程序用到的头文件-----------------
#include <microcontroller.h>//适用于某个型号的微控制器的 C 语
                                                言头文件

//------------定义端口的地址--------------------
#define dataport 0x300//数据输出端口的地址（根据不同微控制器进行
                             设定）
#define ackdryrp 0x301//握手通信信号（ack 信号和 dry 信号）以及
                             rp 信号对应的地址（根据不同的微控制器进
                             行设定）
#define memlim 0x302//存储上限端口的地址
#define MAX 1024//存储空间的最大值，可以自定义，本程序中未使用
unsigned char mem_limit_value; //用来存储设定的上限值

//-----------程序中用到的所有函数和变量-----------------
void get_data (void); //用来从 FSM 获取数据
void send_data_to_FSM (int); //用来向并行端口发送数据
int i;
unsigned char inbyte, outbyte;

//-------主程序---------------
int main (void)
{
 mem_limit_value=255; //设定内存上限值为 255B（可以更改）
 memlim=mem_limit_value; //向 FSM 使用的数据端口发送上限值
 unsigned char array [mem_limit_value]; //C 语言中定义数组
                                                  和位宽
 ackdryrp=0x04; //让信号 rp=1，往图 5.11 里的存储芯片写数据
 get_data (); //用 C 语言里函数的方式来处理发送的数据
 send_data_to_FSM (mem_limit_value); //将波形数据发送给
                                                  FSM 的内存单元
   //此处还可以添加别的命令和函数
 return (0); //C 程序到此结束
}  //主程序结束
```

```
//----------定义主程序中用到的各个函数--------------
void send_data_to_FSM (int mem_limit_value)
{
  for (i=0; i<mem_limit_value; i++)
     {
      do {//等待 FSM 将数据准备完毕的信号拉高
        inbyte=ackdryrp; //操作命令指向端口寄存器 ackdryrp
        inbyte &=0x01; //除了 drdy 信号，其他全部清零
        } while (inbyte != 0x01); //数据准备完毕信号 drdy 有效之
                                   前会一直循环
      //------------------------------------------
      outbyte=array [i]; //从数组中提取下一个发往 FSM 的字节内容
      dataport=outbyte; //将字节数据发送给 FSM
      ackdryrp |=0x02; //将信号 ack 置高并通知 FSM
      do {//等待信号数据准备完毕信号 drdy 被拉低
        inbyte=ackdryrp;
        inbyte &=0x01;
        } while (inbyte != 0x00);
        ackdryrp &=0xfd; //将 ack 信号拉低，提示内存单元写操作
                          结束
      } //for 循环结束
} //向 FSM 发送数据的函数定义结束

void get_data (void)
{ //产生一个锯齿波的数据
  for (i=0; i<mem_limit_value; i++)
     {
       array [i] =i;
     }
}  //获取数据的函数定义结束
```

代码 5.1 波形发生器的 C 代码示例

当然代码 5.1 属于一种广义上的示例，对于特定的微控制器，需要大家专门针对其接口的定义来写专属的代码。

本例主要由一个主函数 main () 和其调用的两个函数组成。其中的 get_data () 函数是用来产生一个简易的锯齿波，方法是通过将字节写入数组，直到存储芯片达到上限值。语句表示为：array [i] =i。

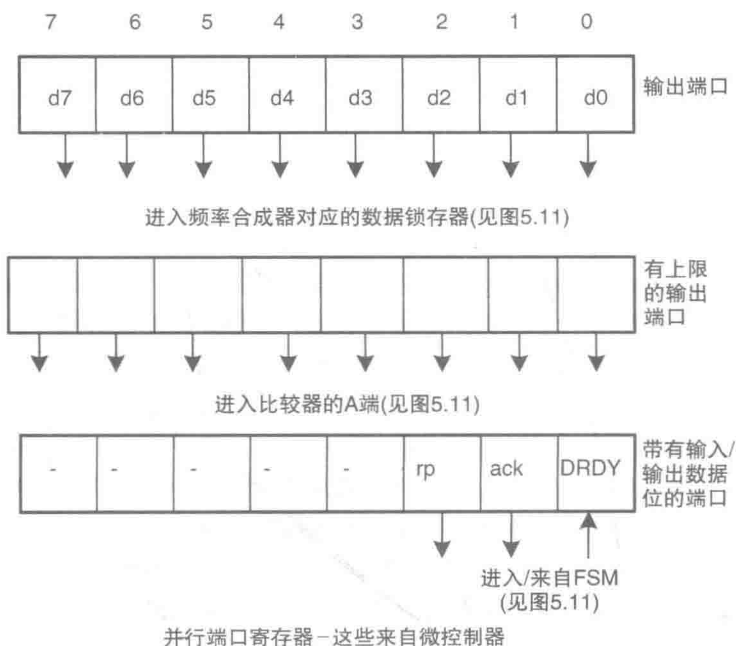


图 5.13　并行端口寄存器及其比特位的定义

for 循环就是将 i 的数值从 0 递增到 men _ limit _ val，并将结果一个接一个地存到数组里去。有必要提醒一下，men _ limit _ val 的值将作为图 5.11 中的比较器的输入端 A，当地址计数器里的值达到上限，即“address limit value”的值和“mem _ limit _ val”相等时，系统将发出 mf 信号。

程序另一个函数的功能是将数组里的内容，通过微控制器的数据端口转存到 FSM 的存储单元里。语句表示为：

```
outbyte = array [i]; //从数组中提取下一个发往 FSM 的字节内容
dataport = outbyte; //将字节数据发送给 FSM
ackdry = 0x02; //将信号 ack 置高通知 FSM
```

drdy 和 ack 被用作握手信号，对上述操作进行控制，并将 FSM 和微控制器进行同步。微控制器的代码用 do－while 循环语句来完成操作。

```
do {//等待 FSM 将数据准备完毕的信号拉高
  inbyte = ackdryrp; //操作命令指向端口寄存器 ackdryrp
  inbyte & = 0x01; //除了字节里代表 drdy 信号的第一位，其余 7 位全
                     部清零
} while (inbyte ! = 0x00); //数据准备完毕信号 drdy 有效之前一直
                             循环
```

上面的 do - while 循环的作用是读取 drdy 信号位的状态值（inbyte = ackdry）。然后用语句 inbyte & =0x01 将 inbyte 里的值和 0x01 按位相与，结果就是除了最低位（bit 0）的 drdy 以外的其他位均被清零。这样的目的是为了和 while 语句所对应的条件 inbyte ！ =0x00 进行比较，当 drdy 信号被拉高后，while 语句内的条件将不再成立，则程序将跳出 do - while 的语句循环。因此只要 drdy = 1 的条件不成立，程序将不会跳出这个 do - while 的循环。第二个 do - while 循环是等待 drdy 信号被清零后，程序才进入下一个提取数据并写入 FSM 中的过程。

程序将不停地重复这些动作，直到所有数组里的数据全部被送到 FSM 的存储单元中。

这部分程序向读者描述了波形图数据是如何被存进 FSM 的整个过程。对于产生更加复杂的波形图数据，例如，正弦波和指数衰减正弦波等，则需要编写更加复杂的 get _ data（）功能函数。

5.6　存储芯片测试系统

基于 FSM 的测试系统，可以用来测试生产线上的存储芯片功能是否完好，这个过程可以在芯片被焊接到 PCB 上之前完成。如果存储芯片有质量隐患而没有及时排查，就被装到 PCB 上，在最终成品质检阶段发现问题，将很难处理。因为此时芯片很难被移除，特别是经过工业化流程焊接的电路板。

存储芯片测试系统可以用在工厂的内部仓储部门，这样每一块存储芯片在入库之后都可以得到及时的检测，方便仓储部门对外部采购的产品进行质量检测和把关。这样的测试系统应该设计成简单易用型的设备，无须操作员学习复杂的测试方法，可以直接通过简单的操作，来判断芯片是否通过测试。

测试方法就是写一些数据到存储芯片，然后再读芯片里的内容，比较是否和写的数据一致来判断芯片质量是否完好。测试过程中，存储芯片的任何单元被发现有故障，都将被判定为质检不过关，从而拒绝入库。

图 5. 14 给出了测试系统的功能框图。这里，十六进制数 55（即二进制 0101 0101）将被用作测试数据写入存储芯片，然后将数据读出，使用数字比较器按位进行比对。按位比较的方法是基于布尔代数公式：Bit*n* = /(A*n*^B*n*)。

这里的^是异或操作符。公式右边是异或非的逻辑运算结果。公式左边的 *n* 代表的是第 *n* 位参与异或非逻辑运算。图 5. 14 后面给出了异或非操作的真值表。

将输入信号 st 拉高后，检测系统被启动。FSM 将控制整个测试过程，并观察输入信号 fab 的值来判断写进存储芯片的值和读出来的是否一致。

更加成熟的设计思路就是检测存储芯片的每一个地址单元，方法是用十六进制数 55 测试完之后，再用十六进制数 AA 重新测试一遍，这样可以检测到相邻比特

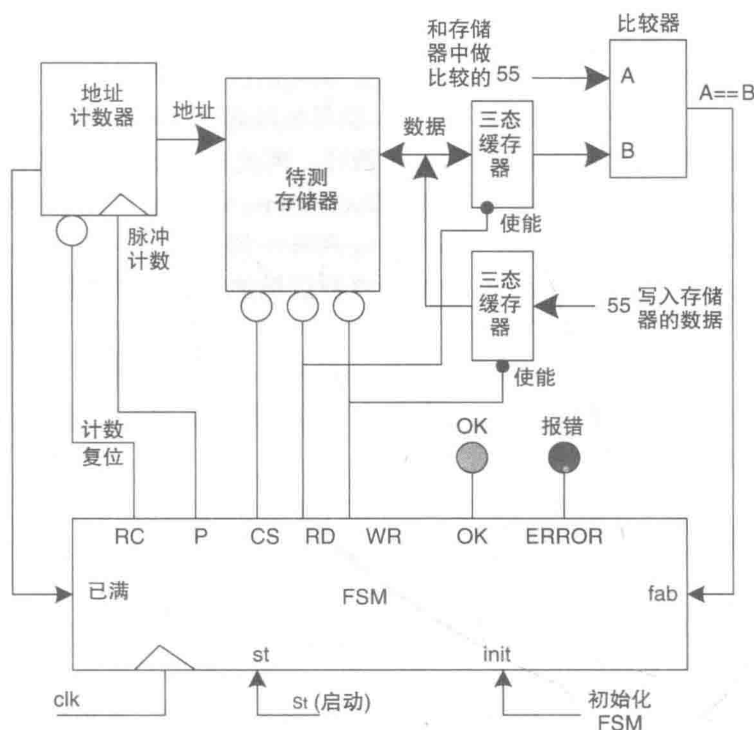


图 5.14　存储芯片测试系统框图

数据位从 0 到 1 或者从 1 到 0 同时翻转时，是否会有被卡住的现象。其他种类的测试，例如检测相邻地址单元数据同时读写是否正常等功能，也可后续逐步加入到系统中去。这里本书只介绍简单的十六进制数 55 的测试。

8bit 的逻辑比较器的输出“A = B”和 FSM 的输入信号 fab 相连。因此如果 bit0 ~ bit7 这 8 个异或非的逻辑输出值都为“1”，那么“A = B”的输出便为逻辑 1。用数学公式可以表示如下：“A = = B” = fab = $\prod_{n=0}^{n=7}$/(A$n^{B}n$)，其中 An 和 Bn 代表了输入端 A 和 B 的第 n 位，然后将每一次/(A$n^{B}n$) 的结果相与（即相乘）。

测试系统的状态框图如图 5.15 所示，输出信号的初始状态在这里没有给出，但是图中每个状态周围都显示了系统各级输出的值的变化。举例来说明，在状态 s0 里，RC = 0，然后到了状态 s1 时，RC 变为 1，并且在其他所有的状态里都没有变化过。同样，在状态 s1 里 CS = 0，因此在状态 s0 时 CS 的值必然是 1。以此类推，其他各个输出在状态 s0 的值为 P = 0，ERROR = 0，OK = 0，W = 1，RD = 1。注意这张状态图的每一个状态都被标注了二次状态变量 ABCD。这些变量在独热系统中其实并不需要用到，但是后面将独热系统和第 4 章提到的常规方法进行比较

时，会起到一定的辅助作用。

在状态 s1、s2 和 s3 里，十六进制数 55 被写到地址计数器指向的地址空间。系统在状态 s4、s5 和 s6 读取地址空间的内容，并在状态 s6 测试信号 fab。如果 fab = 1，则判定地址所对应的存储空间通过测试，FSM 会紧接着在状态 s8 去触发地址计数器指向下一个地址空间。在状态 s9，系统将判断是不是所有的地址空间都被经过测试，如果没有，整个流程将重复循环。

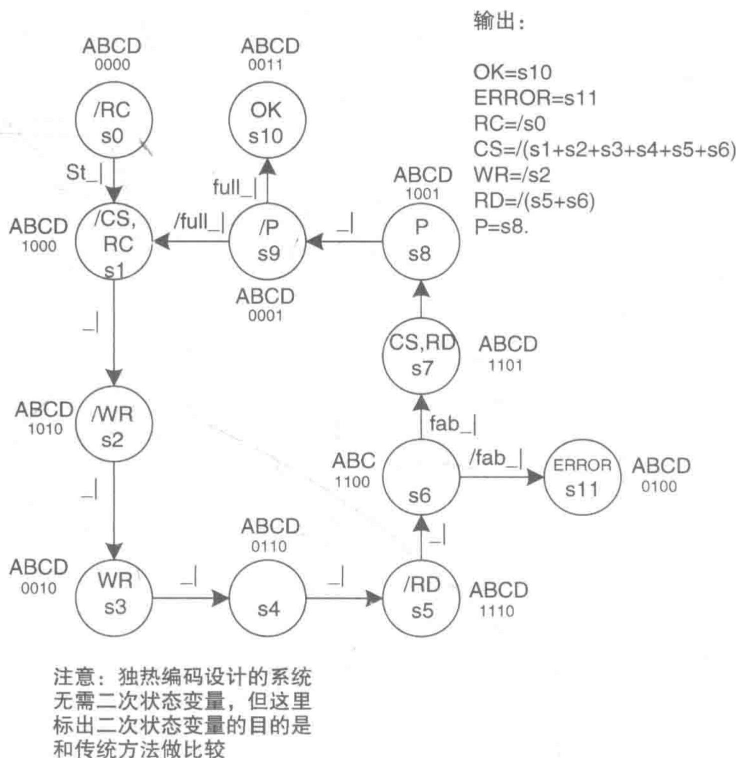


图 5.15　存储芯片测试系统状态图

在测试一个功能正常的存储芯片过程中，FSM 会从状态 s1 ~ s9 不停地循环，直到完成所有的地址空间，接着系统会产生一个信号，迫使 FSM 进入状态 s10。只有通过复位信号才能让系统脱离状态 s10，这样意味着，当测试完一片存储芯片之后，系统将等待操作员的人为介入。

如果任意一个地址空间未通过测试，FSM 会直接跳转到状态 s11 并停止工作。只有通过启动复位信号才能将系统脱离状态 s11。

使用独热编码的系统状态公式如图 5.16 所示。

图 5.15 里中 FSM 有一个摩尔型输出 P。P 的上升沿将触发地址计数器，使得 FSM 进入状态 s8，然后 P 被拉低，FSM 则离开状态 s8。片选信号在 P 的一系列动

采用独热编码设计的公式:
s0·d = s0·/st
s1·d = s0·st+s9·/full
s2·d = s1
s3·d = s2
s4·d = s3
s5·d = s4
s6·d = s5 (由于存在两路分支，而没有保持项)
s7·d = s6·fab
s8·d = s7
s9·d = s8 (由于存在两路分支，而没有保持项)
s10·d = s9·/full + s10
s11·d = s6·/fab + s11

输出:

OK = s10
ERROR=s11
RC=/s0
CS=/(s1+s2+s3+s4+s5+s6)
WR=/s2
RD=/(s5+s6)
P=s8.

图 5.16　存储芯片测试系统独热编码公式表

作之前（状态 s7）会被释放。由于地址计数器只对 P 的上升沿做出回应，因此在状态 s8 之后的下一个时钟的上升沿，系统会在状态 s9 检测内存地址空间有没有被全部测试。

5.7　独热编码和第 4 章常规设计方法的对比

在图 5.15 中，FSM 是带有二次状态变量的，因此需要 4 个触发器来完成。图 5.17 给出了使用 D 触发器来搭建 FSM 所对应的公式。

这样的方法显然是和第 4 章提到的属于同一类型，不属于独热编码。读者可以自己将公式进行简化，并和独热编码公式进行对比。

此时将这两种方法作一个简单的比较是很有意义的。

D触发器设计公式:

A·d = s0·st + s1 + s4 + s5 + s6·fab+ s7 + s9·/full.

B·d = s3 + s4 + s5 + s6 + s11.

C·d = s1 + s2 + s3 + s4 + s9·full.

D·d = s6·fab + s7 + s8 + s9·full.

输出:

OK=s10
ERROR=s11
RC=/s0
CS=/(s1+s2+s3+s4+s5+s6)
WR=/s2
RD=/(s5+s6)
P=s8.

图 5.17　使用 4 个触发器构建存储芯片测试系统

	独热编码	二次状态变量
系统复杂度	简单	需要定义每个状态
触发器数量	12	4
组合逻辑	简单	复杂

独热编码构架简单，虽然使用更多的触发器，但是组合逻辑十分直观。二次状态变量需要一组独立的状态编码，因此就拥有更加复杂的组合逻辑。然而，二次状态变量在上述示例中只需要 4 个触发器和 13 个门电路，独热编码却需要 12 个触发器和 15 个门电路。但要注意到的是独热编码在系统使用 FPGA 芯片的情况下，对于片内空间有更好的利用率。

5.8　动态存储空间访问控制系统

DMA 控制系统一般比较常见的使用场合是计算机主机内部数据交换，或者和其外围设备进行数据交换（例如打印机或者光驱等）。如果这些数据交换的动作是用单片机来操作的，那么整个系统的运行速度将受制于单片机，变得比较慢。一般来说计算机会带有一个专门的芯片称之为 DMA 控制器，例如 8257（现在这类芯片已经被集成到 ASIC 芯片里去了），用来专门做数据搬运的工作。

下面将向大家介绍如何围绕 FSM 设计一个简单的 DMA 控制器。这样的设计原理可以集成到 FPGA 芯片中进行应用。

图 5.18 给出了 DMA 控制器的布局。数据的起始地址、目标地址和发送量（由字节计数器控制）对应的值必须由单片机来提供。由于数据的位宽可以根据实际的设计要求做调节，因此这里可以是字节（8bit）、字（16bit）或者双字（32bit）。本例中，假设上述信息分别是由相应的输入端来提供的，但是 DMA 控制器和这些输入信号之间隔着解码电路，需要控制器通过解码相应的（地址/数据）寄存器来获得其赋值。

图中的虚线范围内表示 DMA 控制器，内存芯片或者外部系统是外接的。

DMA 控制器在不工作的情况下，需要将自身和外接的存储芯片或者其他设备进行隔断，因此在接口部分需要用三态门来实现。

总之，DMA 控制器一开始需要等待一个输入信号 st 来启动。这个时候，它需要收集起始地址、目标地址和传输的数据等信息。然后会发一个中断信号告诉微处理器控制系统将接管存储芯片或者外部设备。微处理器会将自身和这些设备隔断，并发送一个 load 信号告诉 DMA 控制器隔离已经完成，并向 DMA 控制器提供起始地址、目标地址和字节数分别对应的计数器的值。之后，DMA 控制器将把上述 3

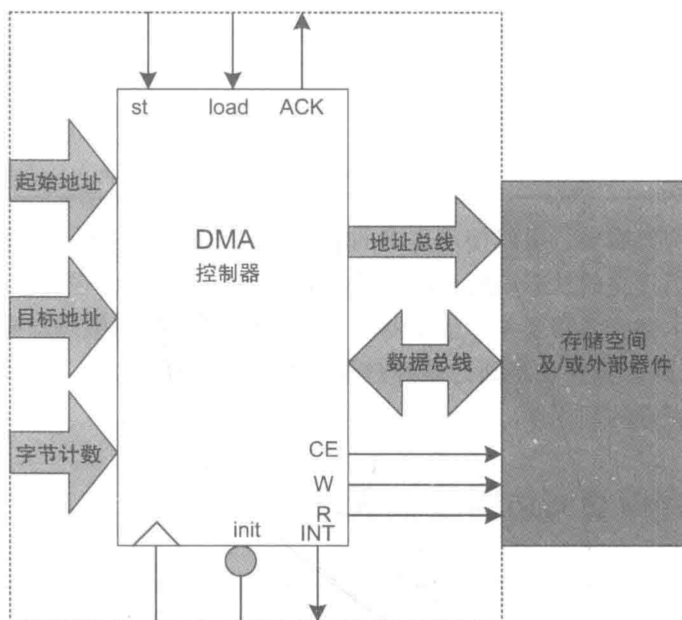


图 5.18 DMA 控制器系统大体框图

个计数器的值导入。

这里要注意的是解码电路是和系统时钟同步的（clk 的下降沿），并且是由 FSM 的输出信号 EC 来控制（使能）。现在 DMA 控制器有了控制整个数据传输所需要的信息。操作步骤包括：

1）选择起始地址计数器，并读取它的值，然后将其存进临时的缓存器。

2）选择目标地址计数器，目标地址计数器中的值存入缓存器。

3）逐个递减字节数计数器的值，并同时逐个递增起始地址计数器和目标地址计数器的值。

4）重复步骤 1~3，直到完成所有的数据传输（即当字节计数器递减到 0）。

有了上述概念之后，现在 DMA 控制器的设计，可以进入到更加具体的环节。既然需要计数，很明显，在两个地址寄存器上需要两个并行加载计数器。同样，字节计数器也是并行加载的，且为递减计数。附录 B 对这些模块有详细描述。

由于起始和目标地址计数器的输出端需要和地址总线相连，因此需要一个三态的缓存器在它们之间将计数器和存储芯片（或者外设）地址总线隔开，以便于在 DMA 控制器不工作时不会占用到地址总线。起始地址计数器和目标地址计数器，在同一时刻只有其中一个在工作，目的是为了避免总线冲突。DMA 控制器也需要连接数据的寄存器和缓存空间，这样便于从一个地址空间向另一个地址空间搬运数

据。数据缓存器在系统中是被用作保存临时数据用的。当不使用的时候，缓存器需要从存储芯片或者外设的数据总线上断开。最后需要指出的是，所有这些内部器件都处于 FSM 的控制之下。

图 5.19 给出实现上述 DMA 控制器的方案框图。图中详细描绘了内部信号的走向，以及它们如何被 FSM 所控制，并完成 DMA 控制器的功能。

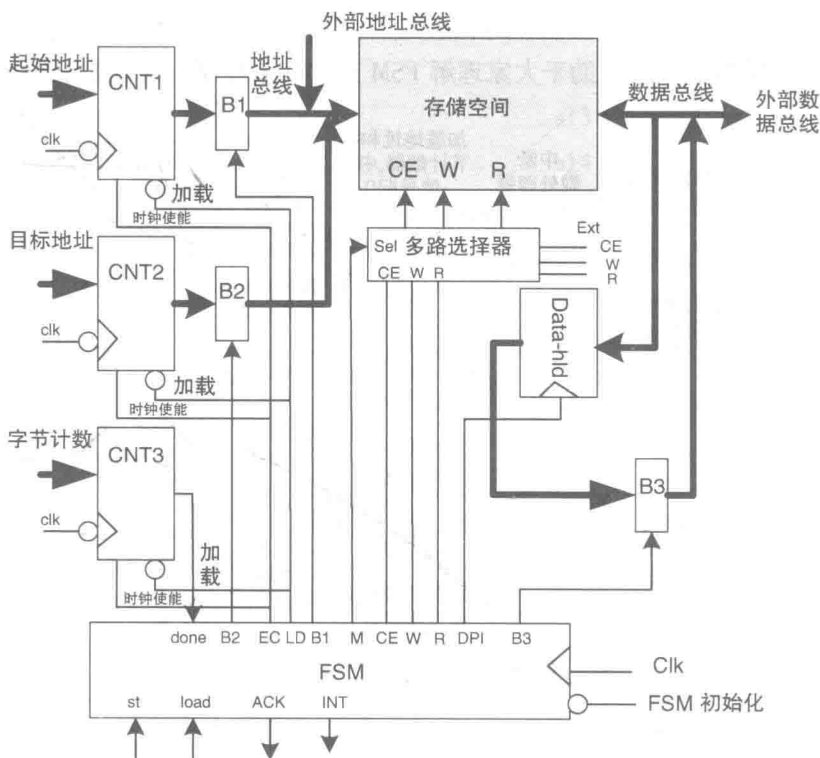


图 5.19 DMA 控制系统详细框图

FSM 需要按照上述 4 个步骤来执行整个功能的控制。这些步骤需要按次序重新被诠释成能够控制图 5.19 中各个硬件部分的具体操作。它们是：

1）等待开始信号 st。

2）发出一个中断信号给微处理器，让其将自身和存储芯片隔离。

3）等待微处理器发出的加载信号；当收到加载信号后，向起始计数器、目标计数器和字节计数器中分别写入相应的值。

4）系统选中起始地址对应的存储空间，并将读取的内容存入数据缓存器。

5）将起始地址从存储空间隔离，然后系统选中目标存储芯片。

6）数据缓存器中的内容此时需要被送往输出缓存器 B3，然后被送往目标地址所对应的存储空间。

7）字节计数器在这个过程中一直被递减，并且系统一直在检测是不是所有的数据都被转到目标地址。

8）如果还有剩余的数据需要传输，FSM 需要将上述 1 ~ 7 的操作再次循环，直到所有的数据传输完毕。此时字节计数器被递减为 0。

现在可以来建立 DMA 控制器的状态图了。图 5. 20 是状态图的最终形态。将此图和图 5. 19 联系起来会有助于大家理解 FSM 是如何来控制 DMA 控制器的。

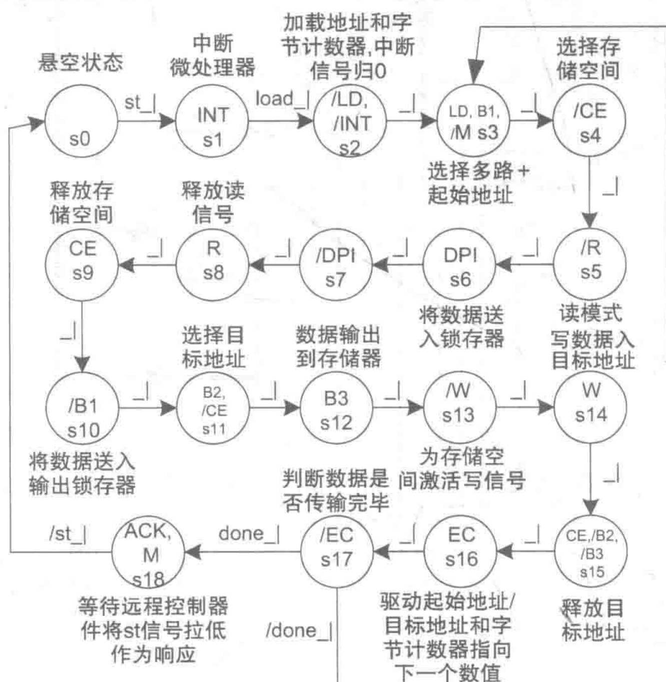


图 5. 20 DMA 控制器状态图

此外还需要注意以下几点：

1）从起始地址读数据的过程中（状态 s4 ~ s7），当把数据向缓存器中传输时（状态 s6 和 s7），片选信号（CE）和读信号（R）需要保持激活状态。这个和之前讨论过的存储芯片读操作有一定的区别。

2）从输出缓存器向存储芯片目标地址写数据的过程看起来更加符合常规：在状态 s11 激活片选，s13 激活写信号；最后在 s14 释放写信号，在 s15 释放片选信号，写数据过程完成。

3）FSM 在状态 s16 通过 EC 信号激活起始、目标和字节计数器，且系统时钟 clk 在其下降沿驱动 3 个计数器进行计数。

这样可以写出独热编码所对应的公式了。

5.8.1　触发器公式

s0 · d = s18 · /st + s0 · /st

s1 · d = s0 · st + s1 · /load

s2 · d = s1 · load

s3 · d = s2 + s17 · /done

s4 · d = s3

s5 · d = s4

s6 · d = s5

s7 · d = s6

s8 · d = s7

s9 · d = s8

s10 · d = s9

s11 · d = s10

s12 · d = s11

s13 · d = s12

s14 · d = s13

s15 · d = s14

s16 · d = s15

s17 · d = s16

s18 · d = s17 · done + s18 · st

5.8.2　输出公式

INT = s1

LD = /s2　低有效信号

B1 = s3 + s4 + s5 + s6 + s7 + s8 + s9

B2 = s11 + s12 + s13 + s14

B3 = s12 + s13 + s14

CE = /(s4 + s5 + s6 + s7 + s8 + s11 + s12 + s13 + s14)　低有效信号

R = /(s5 + s6 + s7)　低有效信号

W = /s13　低有效信号

EC = s16

DPI = s6

M = （s0 + s1 + s2 + s18）　这里用高电平所对应的状态来表示，而不是低电平

ACK = s18

FSM 的仿真如图 5.21 所示。其中，从 s3 ~ s17 所形成的主循环共连续循环了两次。在第二次循环的结尾，输入信号 done 被拉高（逻辑 1），FSM 在回到状态 s0 之前先进入状态 s18，这个与设计的状态图是吻合的。

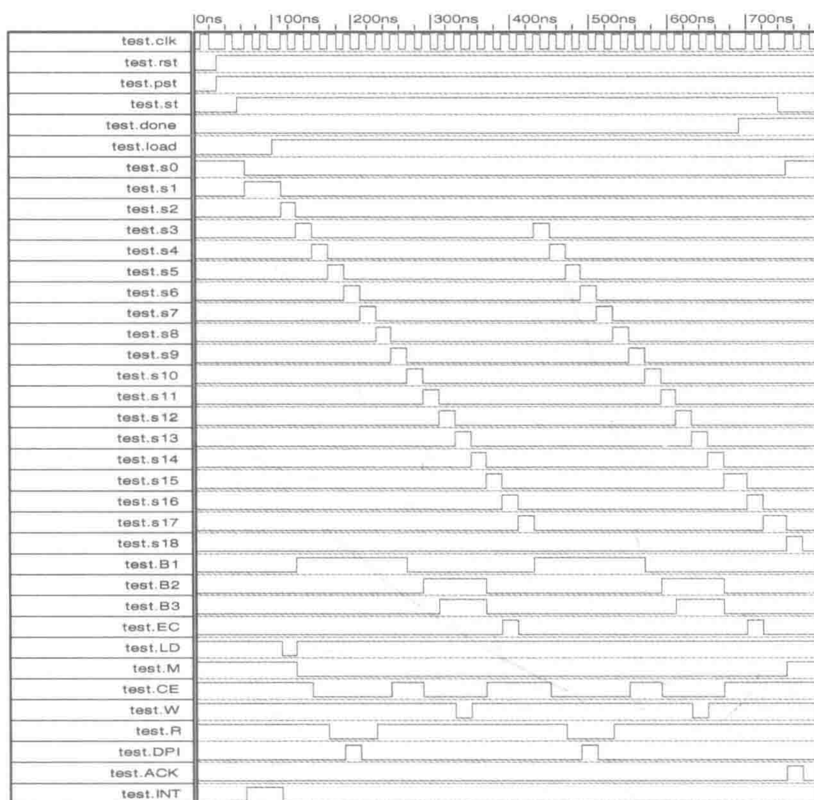


图5.21　DMA控制系统仿真

5.9　如何运用微处理器来控制DMA系统

DMA控制系统在输入信号st激活后就进入工作状态，在上一节中这个开始
号由微处理器的一个输出端送出。某些情况下，用了这种方法就可以省去地址解
逻辑电路了。

更好的办法就是让这个信号从微处理器的存储单元（或者I/O端口）发出
正常情况下，这将占用8bit的位宽。

如图5.22所示，开始信号st由微控制器产生并通过空闲地址输出。地址是
六进制的380或者二进制的11 1000 0000。这是一个典型的内部存储单元的访问
环，当使能信号CE和写信号IOW都有效（低有效，两个信号均由微处理器产生
时，地址解码逻辑电路所对应的地址380h将被置高。在下一个微处理器时钟到
时（T3的上升沿），st信号将被存入D触发器。

微处理器需要通过另一个地址（在这里是386h）在适当的时候输出信号将

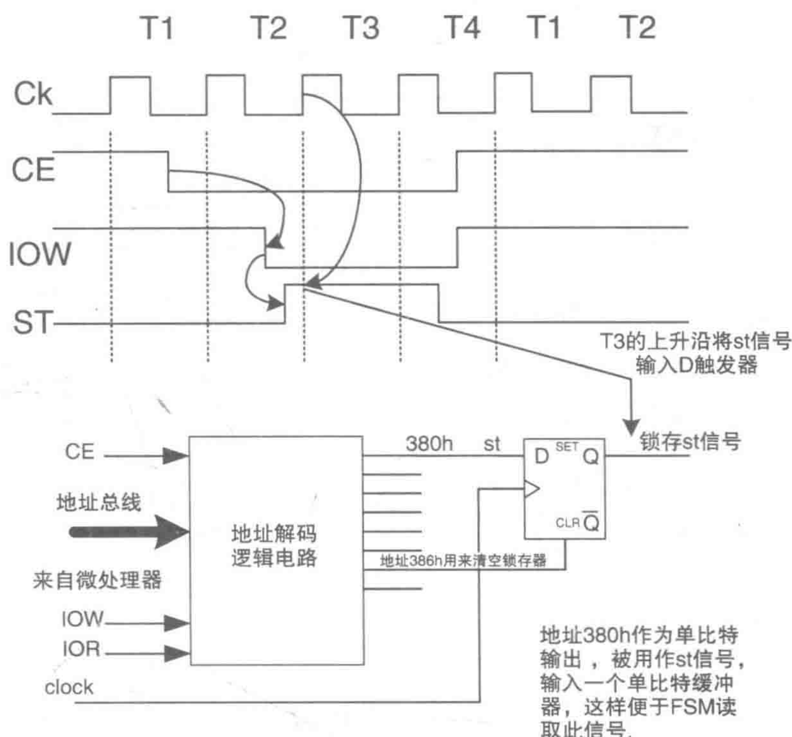


图5.22 用微处理器为FSM产生一个开始信号

触发器重置。然而，在这之前，微处理器还需要等待FSM发出的ACK信号，才能完成上述动作。

图5.23给出了上述过程是如何实现的，同时也囊括了st信号的产生。图5.23左下方的数据锁存器是用来存储FSM发出的ACK信号。FSM在PAK信号到来时，将其用作触发器的时钟，将ACK信号拉高并送入触发器。微处理器可以通过选择地址381h将连接ACK信号D触发器的三态门激活，继而将微处理器数据总线输出端的第0位（bit d0）和ACK信号相连。

ACK信号将在T4时钟周期的上升沿，且CE和IOR信号有效时被读取。之后在信号CE和IOR的上升沿到来时，将ACK信号锁存到微处理器的某个指定的内部存储单元中（见图5.24）。

图5.25给出的状态图，描绘了使用微处理器存储空间或者IO端口映射，来实现上述功能的过程。需要指明的是，这一部分的功能是可以和图5.20的DMA控制器融为一体的。

虽然上述功能设计是基于微处理器总线时序的一些假设来完成的，但是它的确是一种可行的解决方案。

图 5.23　FSM 读写全过程的设计方案

5.10　使用 FSM 检测连续的二进制序列

在通信和计算机网络系统中，一种常用的需求便是检测二进制序列。本节将讨论如何实现这个功能，并且它还可以根据要求进行一些必要的修改，以便探测其他类型的序列。

比较常见的方法就是在传输线上加入一个移位寄存器，然后使用数字比较器来检测放入移位寄存器的二进制码是否正确。这种方法显而易见会带来 $n$bit 的延迟。因此，如果需要探测 4bit 的二进制码，则会产生 4bit 的延迟。如果被检测的数据长度更长（例如 8bit 或者 16bit），或者系统中有其他器件参与其中来检测其他数据，系统的延迟会随之增加。

另一种方法就是实时被动地监控传输路径，并用 FSM 来处理接收到的二进制码字。这种方法不会产生任何延迟。

图 5.26 为系统示意图。其中，FSM 在连续不断地检测输入的二进制编码，并寻找符合 d = 1101 的序列（实际应用中 d 可以是任意组合，这里仅仅是假设）。

FSM 需要将自身和 4bit 数据流的速率同步；图 5.26 中，第一个数据流是

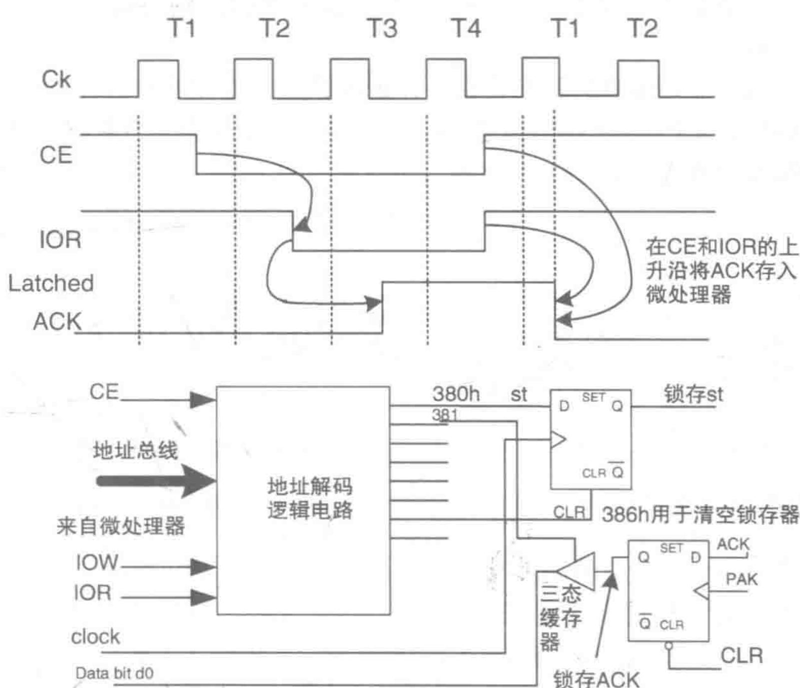


图 5.24　从 FSM 读取 ACK 信号的设计方案

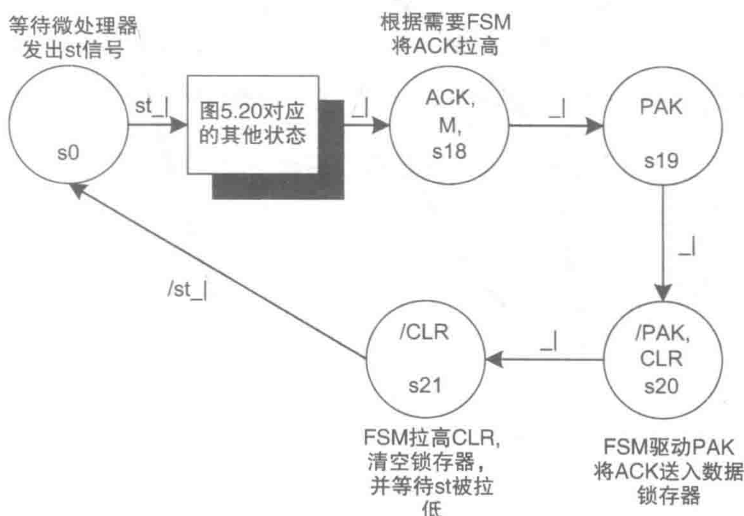


图 5.25　使用微处理器存储空间或者 IO 端口映射的状态图

1011，下一个是 1101（是系统需要的），紧接着是 0011。输出信号 M 必须在 1101 序列出现后置高。

设计状态图需要注意的是不要在一开始就让系统检测到目标序列。如图 5.27 所示，d = 1101 这个序列出现在状态 s4，而不是前几个状态，当系统检测到目标序列时，FSM 会停在那里。

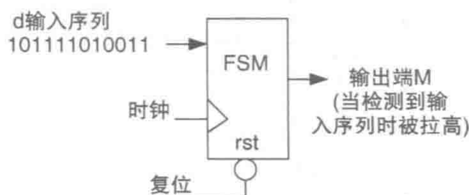


图 5.26　二进制序列检测系统

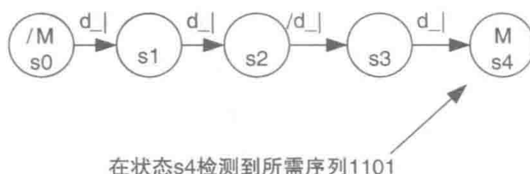


图 5.27　检测目标序列的状态图

然而，系统如果没有检测到目标序列，它需要能够回到状态 s0。这时，图 5.28 所对应的状态图会更加符合要求，它包含了所有可能的情况。

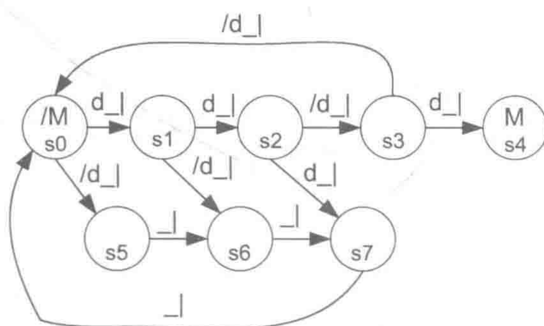


图 5.28　检测目标序列的状态图（完整版）

例如，当输入序列 d = 1100 时，FSM 的走向应该是 s0，s1，s2，s3，s0。如果输入序列是 1111，FSM 所经过的状态路线则为 s0，s1，s2，s7，s0 等。用这样的方法，FSM 就能够紧跟输入序列的步伐，做到实时监测。

一旦监测到目标序列（这里为 1101），FSM 会停在 s4。

FSM 的采样时钟必须对准被检测数据中央。具体方法可以参照 4.7 节所介绍的串行异步接收系统来完成（见图 4.20）。

独热编码也可以在这里得到应用，具体公式如下：

$s0 \cdot d = s3 \cdot /d + s7$

$s1 \cdot d = s0 \cdot d$

$s2 \cdot d = s1 \cdot d$

$s3 \cdot d = s2 \cdot /d$

$s4 \cdot d = s3 \cdot d + s4$

$s5 \cdot d = s0 \cdot /d$

s6 · d = s5 + s1 · /d

s7 · d = s6 + s2 · d

输出公式则为：M = s4。

系统设计也可以用 Verilog HDL 来完成，图 5.29 为其仿真结果。图中仿真了所有可能的状态，目的是检测 FSM 是否符合设计初衷。

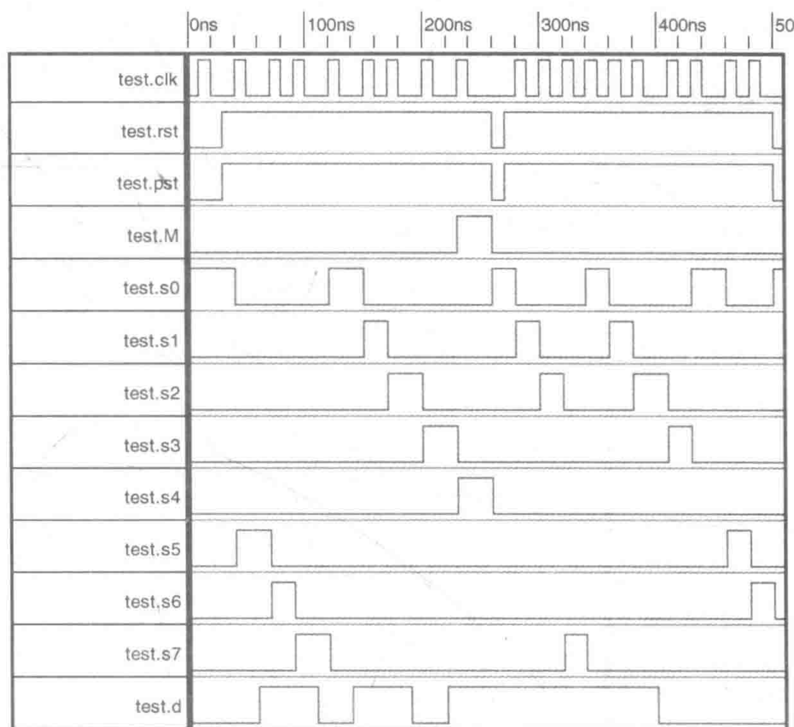


图 5.29　序列检测仿真图

一开始，仿真所对应的状态走向是 s0，s5，s6，s7，s0。然后是 s0，s1，s2，s3，s4，紧接着 M = 1。之后 FSM 会将 rst 和 pst（异步初始化信号）拉低，系统复位到 s0，开始新的一轮检测。s0，s1，s2，s7，s0 是复位后 FSM 的第一个走向，然后还有其他的状态组合，显现出一个完整的仿真结果。注意到在最后（即 s0，s5，s6），异步初始化信号被激活，强行将还在运行的 FSM 复位到 s0。

系统还可以进一步地被优化，让其不间断地检测输入的信号，一旦目标序列被检测到，则输出 M = 1。方法很简单，只要将 M 在状态 s3 改为米利（Mealy）型输出，这样 M = s3 · d。

如果进入状态 s3 以后，下一个 d 不是 1，则 M 也不会为 1。这样系统就不再需要状态 s4 了。

图 5.30 是修改后的状态图。图中，M 被改为米利（Mealy）型输出，合并到状态 s3 中，这样系统无论 d 为 0 或是 1 都能回到 s0，而且此时的系统便能以 4bit 为单位一直不停地检测下去。

在图 5.31 中，仿真结果显示系统在检测到序列 1101 以后从状态 s3 回到 s0。注意输出 M 只有在 d = 1 时才为 1。

同样的方法可用于检测更长的序列，区别只在于需要使用更多的状态和更多的触发器。

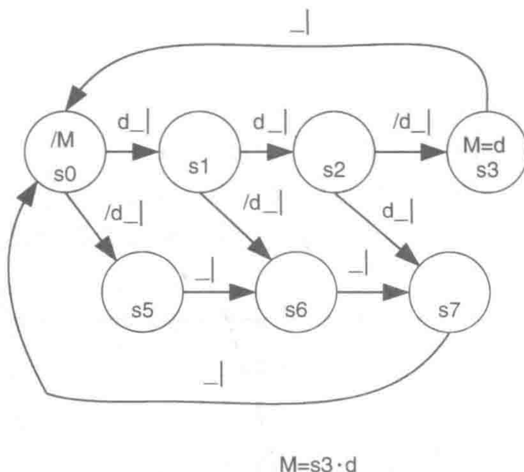


图 5.30　连续检测序列 d = 1101 的最终状态图

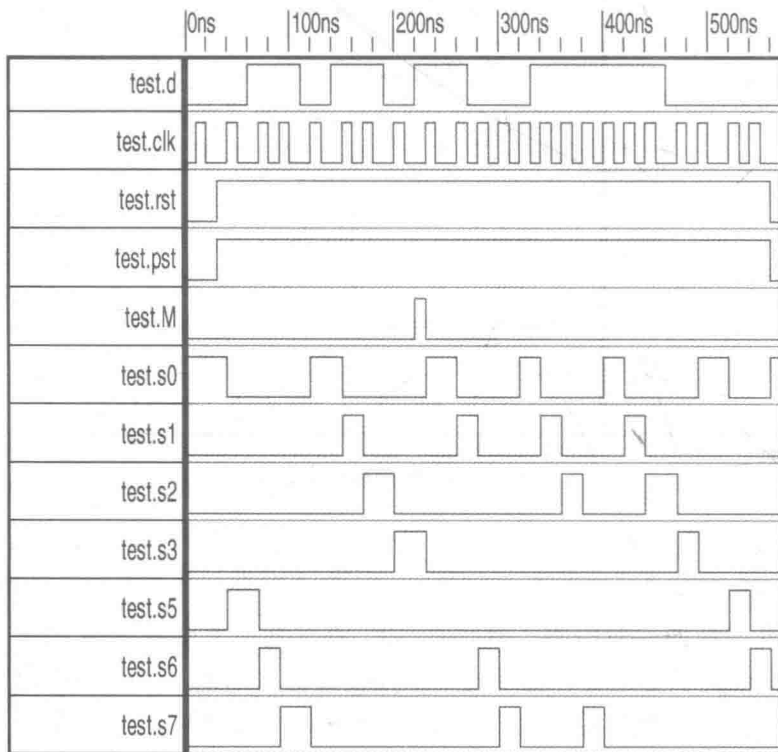


图 5.31　序列检测最终仿真结果

但是像图 5.30 所描绘的系统也有一个局限性，就是被检测序列的值必须是固定的，在这里，目标序列是 1101，不能变化。如果有一种 FSM 能够检测任意二进

制序列，而不需要重新设计状态图，将会十分地实用。

针对刚才所说的局限性，可以将输入信号d和一组比较器逐位比较（同或门），比较器的每一位和输入信号的每一位进行一一对应比较，如图5.32所示。这种情况下，可以实现8bit数据的检测。

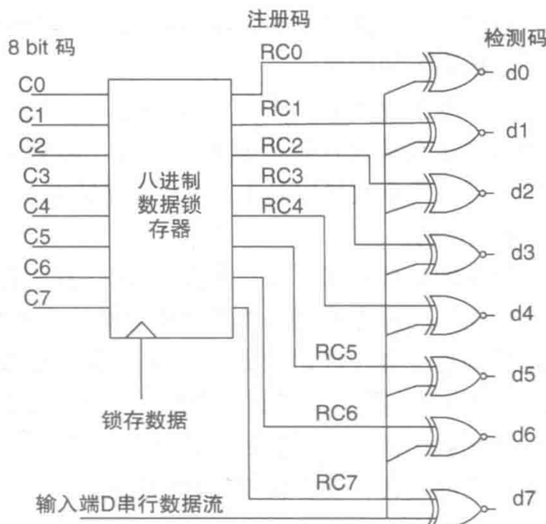


图5.32 使用比较器和预存的每一位数据进行比较

同时，我们在检测数据之前，可以将输入数据放入一个锁存器。这种设计方案可以支持最多255种不同的二进制码的检测（假设0000 0000不在被检测范围内）。

二进制码C0～C7经过数据锁存器输出后，以预存数据RC0～RC7的形式作为比较器的输入。数据信号d始终作为比较器的另一个输入，比较结果d0～d7作为输入进入FSM进行处理。

图5.33描绘了整个系统的构架：FSM需要一个额外的使能输入信号en去激活整个流程。此外FSM还需要输出一个信号LD去锁存被检测的数据。

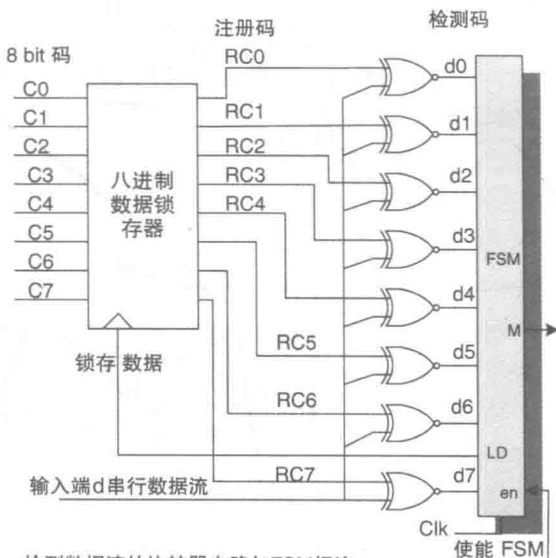


图5.33 通用8bit二进制码检测系统

系统所对应的状态图如图5.34所示。它的基本思路和图5.30类似，只是位宽变成了一个字节（8bit）。注意这里的主要区别在于FSM不再实时地比较输入信号d的值，而是直接接收比较器的比较结果，先是bit d0，然后bit d1，bit d2直到bit d7。

这样FSM的工作变得比较固定，只要在比较前将被检测的8位数据加载到锁存器中即可，而且现在系统能够比较任意8bit的数据序列。信号en可以在任何时间停止系统运行，但是当en被释放后，当前正在进行的检测过程仍将完成，然后系统将回到s0。

试想一下，如果在锁存器的输入端不停地变换码字，那么 FSM 就能够一个接一个地检测不同的码字。

关于本系统还有一个问题没讨论，就是如何将 FSM 和输入的数据序列同步。一种方法是在系统开始检测之前预存一段数据，例如 10101010 × ×，其中额外的两位 × 可以是 0 也可以是 1。并且预存数据是提前告知发送端（数据 d）和接收端（锁存器）的。

附加的 2 位数据可以让 FSM 向锁存器中存入系统真正要检测的数据。这里同步的概念是，首先将预存的数据作为被检测数据，装载到锁存器中进行比较，同时发送数据端也会在数据线 d 上发送带有预存数据的序列，一旦预存的数据被检测到，即 M = 1，则实际需要被检测的数据，会在 FSM 回到下一轮检测开始时（状态 s2）被装载进锁存器，然后系统进入正式的检测循环。因此这里的同步实际上是一个预检测的过程，或者说初始化的过程。

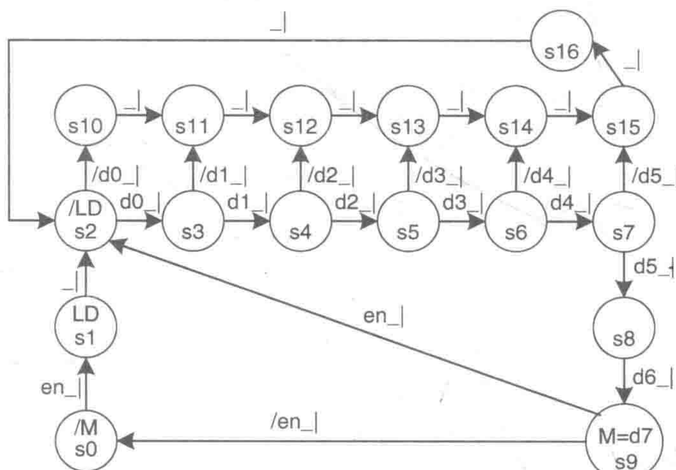


图 5. 34　基于 FSM 的单字节序列检测系统状态图

基于图 5. 34 的独热编码公式如下：

s0 · d = s9 · /en + s0 · /en

s1 · d = s0 · en

s2 · d = s1 + s9 · en + s16

s3 · d = s2 · d0

s4 · d = s3 · d1

s5 · d = s4 · d2

s6 · d = s5 · d3

s7 · d = s6 · d4

s8 · d = s7 · d5

s9 · d = s8 · d6

s10 · d = s2 · /d0

s11 · d = s3 · /d1 + s10

s12 · d = s4 · /d2 + s11

s13 · d = s5 · /d3 + s12

s14 · d = s6 · /d4 + s13

s15 · d = s7 · /d5 + s14

s16 · d = s15

输出为 M = s9 · d7 LD = s1。

当然，被检测数据的位数可以增加到任意长度，相应的状态图也可以基于这个方法设计。图 5.35 所示的仿真图所对应的被检测序列为 11001011。这段序列首先

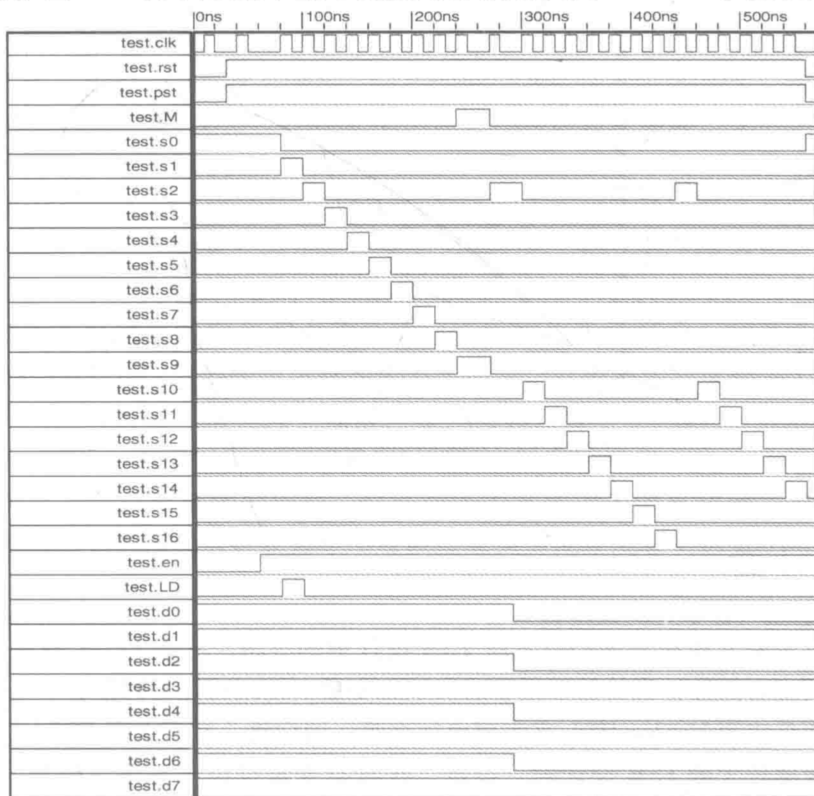


图 5.35 使用数据 11001011 仿真序列检测系统 FSM 的结果

被装载到锁存器，然后通过 d 输入一串数据，发送端在数据流的最后发送需要被检测的序列。M 在系统检测到被测数据后被置高。

完整系统的仿真如图 5.36 所示，其中包括比较器、锁存器以及 FSM。图中涵盖了图 5.33 和图 5.34 所对应的输入、输出和所有状态。可以看出，在大概 700ns 的地方，系统检测到序列 11001011。

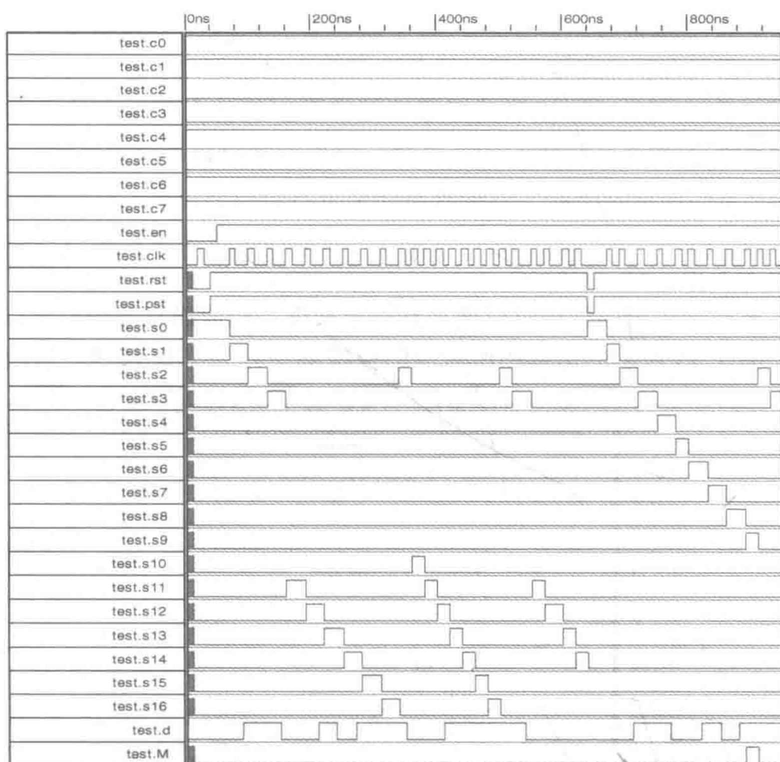


图 5.36 完整的 8bit 序列检测仿真

5.11 小结

本章主要介绍独热编码技术在 FSM 设计中的应用，它在 FPGA 等器件的片上设计中所拥有的独特优势是不需要二次状态变量。使用独热编码推导状态公式非常便捷，而且应用 Verilog HDL 来实现也很直观。同样地，对于大型 FPGA 芯片来说在遇到规模较大的 FSM 设计时，设定二次状态变量也由于独热编码技术的存在而变得不再那么的必要。

第 6 章　Verilog HDL

6.1　硬件描述语言背景介绍

本章将开始介绍当前数字系统设计者使用的主流设计工具——硬件描述语言的一些基本知识。根据不同的设计目的，会有很多种硬件描述语言存在。其中有一些针对底层的设计，运用的是逻辑门和布尔代数（例如 ABEL[1]）；而其余的一些则是所谓的系统级描述语言，主要作用是辅助设计和整个系统硬件与软件功能的验证（例如 System C[2] 和 System Verilog[3]）。

除了针对事件和赋值的单个系统的辅助设计，硬件描述语言也已经逐步地升级到可以处理随着时间连续变化的模拟信号的领域。若非必要，本书将不对这类语言做特别详细介绍。

在此向大家介绍的 Verilog HDL[4] 目前运用广泛，且相对容易学习，很多场合直接被称为“Verilog”或者“HDL”（“Verilog”或者“Verilog HDL”这两个名字会在本书中交替出现）。目前 Verilog HDL 在数字系统设计领域，无论是工业应用还是学校教学，全球范围内都有相当数量的一批固定用户。对于描述和仿真数字系统，Verilog HDL 有自己独立的一套辅助系统。由于使用系统内嵌的 MOS（金属氧化物半导体）晶体管模型，Verilog HDL 可以基于所谓的“开关级”来设计数字电路，这样每一个开关电路上的时序和信号强度都可以用行为的方式进行描述。通俗来说，开关级电路和数字集成电路的物理特性变化非常接近，这就使得 Verilog HDL 变成在验证设计方面超越电路图的首选语言。从另一个角度来说，在使用更加抽象和功能强大的“行为级”或者“寄存器转换级”（Register Transfer Level，RTL）的电路设计系统时，面对存储单元和信息处理等操作，使用 Verilog 里的高级语言构架能够获得更多的便利。也许是因为其强大的语言功能，使得 Verilog HDL 和其他类似的编程语言成为唯一可以应对当前复杂度较高的数字电路设计的工具。

在 20 世纪 80 年代，Verilog HDL 一开始只是一种内部设计工具，但随着数字电路和系统设计的复杂度逐步提高，它很快就得到了广泛的关注。随后，它被普及并在 20 世纪 90 年代中期被列入 IEEE 标准。本书中大部分工程案例使用的 Verilog HDL 是基于 2002 年推出的代号为 IEEE1364 - 2001 标准来定义的。这个版本的 Verilog HDL 新增了许多功能强大的特性，经过一系列的改良，让它和广为人知的硬件描述语言 VHDL（高速集成电路硬件描述语言）许多地方看起来有相似之处[5]。

这两种硬件描述语言，有相同的英文缩写（Verilog HDL 和 VHDL），在语法和整体表现形式上却有所不同，VHDL 和 Ada 编程语言[6]看起来相似，而 Verilog HDL 里面带有一些 C 语言的元素。除去这些外在的差别，这两种硬件描述语言在语义的描述方面有很多相似的地方，对于同样的设计目的和对性能以及成本有要求的数字电路系统，两者的运用方式也基本相同。

对于设计的仿真和验证，这两种语言均可以生成被称为“逻辑综合”[7]的自动硬件集成系统的源文件。目前大量主流数字电路的硬件部分，都是基于它们其中一种语言的设计描述综合而来。眼下逻辑综合的工具软件都有很高的可靠性，并且通常能够以可编程逻辑电路的形式优化设计方案，提高设计效率。尽管有这类工具存在，但还要指出的是数字系统工程师的职责仍然很重要。与过去不同的是，设计师们现在可以处在一个更高层次的、抽象的平台，利用硬件描述语言强大的表达功能进行更加复杂的设计，大部分细节问题和流程化的执行步骤已经被自动化了。

如今设计语言的运用平台已经成熟了，因此需要当代设计师至少掌握其中一种硬件描述语言来满足招聘市场的需求。一旦学习了基本原理，将设计思路从一种硬件描述语言移植到另一种语言平台很方便。和已经掌握一门硬件描述语言，仅需要将已有设计转换到另一个语言平台相比，从零开始学习和掌握一种硬件描述语言并能够将其运用到设计工作中更具备挑战性。

就像之前提到的，由于主流硬件描述语言的快速发展，加上硬件和软件设计的集成度、复杂度的不断提高，导致了微电子技术的日新月异，并由此派生出所谓的系统级语言，例如 System C[2] 和 System Verilog[3]。

由于 System C 是基于 C + + 语言发展而来的，因此在数字电路设计领域缺乏底层技术支持。而 System Verilog 是 Verilog HDL 的一种扩展形式，除了拥有 Verilog HDL 所有的硬件设计模块功能以外，还具备现在流行的“片上系统”（System Of Chip，SOC）设计所要求的高级数据提取和软件集成等特性。

通过学习 Verilog HDL，工程师们不光掌握了可以应对更加复杂的编程语言，还可以学习工具软件的使用，拥有这样的优势就可以为自己规划一条长期的、前景广阔的职业道路。

现在来总结一下使用像 Verilog HDL 这样的硬件描述语言的一些优势：

1）独立设计——用硬件描述语言编写的程序，基本可以独立于产品本身的设计限制，因此适应性很广泛。

2）编程语言简明、清晰，并有独立的文件生成。

3）标准的语言构架支持设计方案复用，并能衔接不同的设计工具。

4）程序描述具有替换或者补充说明原理图的功能。

5）自动化设计——逻辑综合工具可以接纳用硬件描述语言设计的方案。

6）高级设计——设计者可以从枯燥的门级设计中解放出来，从而转向系统级设计。

6.2　用 Verilog HDL 进行硬件建模：模块

在 Verilog HDL 里，最基本的硬件单元被称之为模块。和 C 语言一样，每个模块都是独立的，在定义和功能上不能够和其他模块重叠或者嵌套。一个模块可以被另一个模块调用，就好比一个 C 语言函数可以被另一个 C 语言函数调用一样。这些特性构成了 Verilog 语言的设计构架。

代码 6.1 给出一个基本模块的格式：

```
module module - name (list - of - ports);
local wire/reg declarations
parallel statements
endmodule
```

代码 6.1　模块的基本格式

需要注意的是这里以及后面所有的代码里，关键字都会用粗体字标识出来。硬件描述语言的主体部分是用 module 和 endmodule 来封装的，前一个后面会立即跟上模块的名字和端口名称的列表，所有端口名称必须写在括号里（某些模块不需要端口，因此也就没有端口列表）。

在端口列表的末尾，分号“;”总是必须出现的，而在模块的结尾，也就是 endmodule 后面却不需要分号。

在模块的开头，括号里的端口列表必须明确每个端口的大小（信号的位宽）和方向（输入还是输出等），以及每个端口的名称。

这种格式，使得模块的第一行包含了所有的输入输出的端口信息，这些信息从模块外部也是可见的，即模块的标题代表了模块的界面或者模块的原型。

模块标题的下面列出了一些模块内部使用的变量。代码 6.1 的第二行声明了 Verilog 里最常用的局部对象变量类型：reg（寄存器型）和 wire（线网型）；它们分别代表了模块内部存储功能或者内部信号之间的连接。和其他类型的语言一样，Verilog 要求所有对象在使用前必须被声明；因此，这意味着定义它们的语句应该出现在模块的开始部分。局部 wire 型和 reg 型对象代表了模块内部的信号类型，并且它们成为描述逻辑单元之间互联关系的所谓并行语句的一部分。这里的术语“并行语句”代表了这部分声明语句在仿真时的执行方式，即同时被执行，跟实际的数字硬件电路有点类似。并行语句描述了模块内部硬件电路的行为、结构或数据流。语句的形式是多种多样的，可以是原始门电路、模块建模和连续赋值语句等，所有这些将在后续的章节向大家逐个介绍。

Verilog 模块里的语言格式是 ASCII 文本，如果模块的名称为 module - name（模块名），则系统默认的存储文件名则为“module - name. v”。图 6.1 给出了一个十分简单的模块，图中的语句每一行开头还带有数字标号（行标）以供参考引用，

行标在实际源文件里是不出现的。

如图 6.1 所示，模块描述的是一个两路输入的异或门，名字为 myxor，输入 a 和 b 都是单比特输入，y 为单路输出。模块里信号的名称和方向在第一行的模块名后面的括号里体现，并用逗号隔开。Verilog 语言里信号默认的是一位（1bit），有时候，某个信号可能既作为输入也作为输出，即双向。Verilog 使用一个专门的词 inout 来表示双向端口。

图 6.1 模块的功能在程序的第二行，用连续赋值语句来体现的，表达式 a^b（其中符号^在 Verilog 里表示逐位异或）赋值给输出信号 y。这里关键词 assign 的意思是连续赋值，它表示操作符“ = ”的左右两边的关系是恒定不变的。因此，它一般用来描述组合逻辑。

```
1 module myxor(output y, input a, b);
2 assign y = a ^ b;
3 endmodule
```

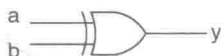


图 6.1　一个简单的模块

除了和 C 语言里的赋值语句比较类似以外，图 6.1 里第二行的连续赋值语句同时也是一个并行语句；这表明此语句永远有效，并等待输入信号 a 和 b 的任意一个发生变化，然后触发公式得到新的输出。这种变化依赖于模块所连接的外部输入信号的变化情况。

具体来说，无论 a 和 b 的其中一个或是两个信号均发生变化，都将触发操作符“ = ”右边的表达式更新输出的结果，这个结果将在下一个仿真周期的开始赋值给输出信号 y。

一个模块可以含有多个连续赋值的声明语句，所有语句都可以同时执行，因此在用键盘输入语句时顺序是不分先后的。

图 6.2 为一个含有多个连续赋值语句的模块。类似代码风格有时被称为数据流格式的描述。代码下方是对应的逻辑门电路图。本例中，每一个逻辑门都用一行独立的连续赋值语句来表达（代码的第 7、第 8 和第 9 行）。另一种表达方式就是只用一行语句来表达整个异或门的逻辑关系，例如

```
assign F = ~((A & B)|(C & D));
```

上述语句显现出 Verilog 和 C 语言在逐位逻辑操作上的相似性，例如非门（~），逻辑与（&）和逻辑或（|）。同时大家需要注意上述语句里括号的使用，逻辑运算的优先级在括号的作用下才能得出正确的结果。

代码第 7、8、9 三行的连续赋值语句的顺序是可以任意变换的，这个不会改变逻辑门的运算结果，具体行为描述为内部单比特线网型（wire）信号（在第 6 行定义的 W1 和 W2）作为两个与门的输出（第 7、8 行），同时又作为后续或非门的两个输入。它们的执行是由外部 4 个主要输入信号 A、B、C、D 和内部线网型信号 W1 和 W2 来决定的。

例如，如果输入信号 A 的值从 0 变为 1，此事件将触发第 7 行的语句执行。假设输入 B 的值已经是逻辑 1，紧接着，输入 A 的变化将导致信号 W1 的值从 0 变为 1。需要说明的是信号 W1 上的变化和输入信号 A 上的变化是同时发生的，因为连续赋值语句不会产生任何传输延迟。然而，仿真器在更新信号的赋值时，使用的是一种称为仿真循环的独立循环机制，其中信号的值只能在语句执行后才被更新。

每出现一个仿真循环，就会形成一个非常微小的延迟，通常被称为延迟增量。所以，如果输入 A 上面的变化在 10ns 时出现，则连线信号 W1 上的变化就会在 10ns + 1d 的时候产生，其中 d 就是我们所说的增量（delta）。

再看图 6.2，W1 上面的触发事件会触发第 9 行命令的执行，当然最终输出信号 F 是否出现变化还要取决于 W2 上面的赋值。如果输出信号 F 的值被更新，那么更新的时间点则会是 10ns + 2d，因为每一条语句的执行都会产生一个延迟增量。

线网型信号 wire 在 Verilog 语言里是一种比较典型的通用对象，也称为网络，所有的网络信号都需要被连续驱动，驱动来源可以是连续赋值语句，可以是逻辑门的输出信号，还可以是整个模块的输出等。

注意：连续赋值语句的左边，或者说是被赋值对象必须是线网型（wire）变量。

模块的端口默认也是线网类型的（如图 6.2 中第 5 行描述的 A、B、C、D 和 F）；它们本身出现在连续赋值语句的左边还是右边，取决于被定义为输入还是输出。不像一些其他的硬件描述语言，Verilog HDL 是允许输出信号出现在操作符的右边的。这种灵活多变的特性反映出硬件设计中存在着一种比较普遍的现象，即模块的输出作为反馈信号输入到模块本身。

图 6.2 同时还告诉我们如何在 Verilog 语言里添加代码注释。第 1、2、3 行介绍了 Verilog 中是如何注释的，其格式和 C 以及 C + + 语言相似。语句注释在描述

```
1   // 注意—注释的书写
2   // 格式和C++类似(多行语句的注释用/* */)
3   // 用Verilog描述1个与门或非门
4
5     module AOI(input A, B, C, D, output F);

6     wire  W1, W2;

7     assign W1 = A & B;
8     assign W2 = C & D;
9     assign F = ~( W1 | W2);

10   endmodule
```

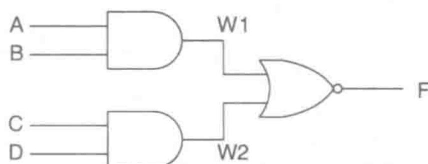


图 6.2　用 Verilog 描述与 - 或 - 非模块

设计思路上是很有用的辅助文档。

下面将介绍 Verilog 模块的一些其他的新功能。

图 6.3 给出一个简单的 4 位二进制加法器的模块框图和代码。模块的标题占用了前 4 行代码，并显示如何定义多位信号。本例中，输入信号 a 和 b 以及输出 sum 都是四位的位宽（4bit)，以总线的格式标记。

```
1  module  add4( output  [3:0] sum,
2                output  c_out,
3                input  [3:0] a, b,
4                input  c_in);

5  assign  #15 {c_out, sum} = a + b + c_in;

6  endmodule
```

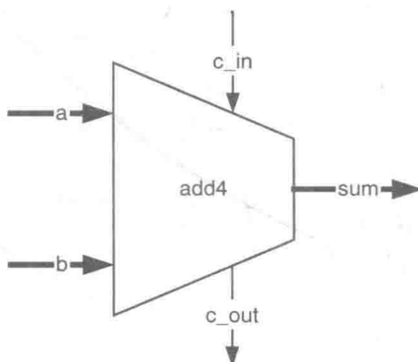


图 6.3　4 位加法器 Verilog 代码和模块框图

图 6.3 里代码第 3 行定义了两个输入信号分别是 4 比特（4bit）位宽，排序为 3 ~ 0

```
input [3: 0] a, b,
```

具有相同方向和位宽的端口可以在同一行列出，用逗号隔开，但是推荐不同的信号分开表述，比较清晰。这里的［3:0］是指端口的位宽；为方便计算，最左边的一位（这里指 bit3）总是默认为最高位。

图 6.3 的模块的具体功能描述只有一条语句，就是在第 5 行：

```
assign #15 {c_ out, sum} = a + b + c_ in;
```

这条语句充分体现了像 Verilog 这一类硬件描述语言的强大的表达能力。要描述一个加法器，最简单的方法就是用内嵌的加法操作符“+”，将三个输入信号的值相加，然后将结果实时地送到输出端。根据需要，还可以用布尔代数公式、逻辑门，甚至单个的 MOS 晶体管来描述加法器，这也体现了 Verilog 的灵活性。关于上述语句有几个要点需要在这里强调一下：

- 操作符右边的表达式默认执行的是无符号的加法。

- 式子里没有标明 a 和 b 的哪些位参与运算，表明两个信号每一位都参于和单比特进位输入信号 c_ in 相加。
- 进位位 c_ in 会和 a、b 的最低位相加（即 a [0] 和 b [0]）。
- 加法器的结果最多有可能是 5 位；因此，连续赋值语句的对象是输出信号 c_out和 sum 的组合（用操作符 {} 来体现）。其中 c_out 占据了最高位（bit 4）。
- 在关键词 assign 后面加了#15，代表输入信号的变化和输出信号的变化之间有 15 个时钟单位的延迟。关于时钟延迟将在第 7 章有更详细的介绍。

上述 Verilog 模块中，所有信号属性均为线网型（wire），这其中包含了内部信号和模块端口信号。这是由组合逻辑的特性决定的，首先每一个模块的输入和输出之间都有固定的组合逻辑关系；其次模块不需要存储任何数据。断开的线网信号会立刻失效，之后的值呈现高阻 z。

除了线网型（wire），Verilog 还提供了寄存器型（reg）变量，用于描述能够保持或存储某个数值的信号类型，一般看到的都是系统最后赋予它们的数值。

和线网型（wire）类似，寄存器型（reg）默认也是 1 位（1bit），但也可以将其定义成多位的总线型，和上面的线网型（wire）的定义方法一样：

```
reg [7: 0] count; //一个 8 比特寄存器变量
```

关于寄存器型（reg）对象的使用将在第 7 章详细介绍。

6.3 模块的嵌套：建立构架

设计过程中用到的最重要的一种设计思路便是自顶而下的设计方案。简单说来，就是将一个较为复杂的完整系统拆分成一些独立的分系统，每个分系统根据情况还可以进一步细分为更加简单的小模块。在 C/C + + 和其他类似的语言里，最基本的执行单元是函数和进程；这些独立的代码都是在执行一个相对简单的任务。软件工程师首先编写这些能够执行简单任务的函数，然后在顶层通过调用的方式来使用它们。通过这种方式，一个复杂的软件应用程序形成了由多个函数组成的构架，并且可以多重嵌套。在数字硬件设计中，类似构架可以通过模块的建模来实现。

根据之前的描述，用户可以利用现成的模块或者预设的逻辑门、开关电路进行建模。通过这种方法，Verilog 为复杂数字系统的设计提供了有效的支撑，即层次化、模块化和统一化[8]。

用 Verilog 构建分层的系统设计方法很简单。首先建立模块源文件并以文本格式保存，然后源文件可以在标准库（或者在工具库、数据库）里被编译，接着就可以在其他模块里被调用，格式如下：`Module - name instance - name (list - of - connections)`。

上述语句里，module - name 是模块标题定义的模块名，instance - name 是一个针对此模块的特定情况或事件而设定的独立名称。List - of - connections 定义了模

块的端口在模块内部或者与上一级模块之间是如何关联的。

图 6.4 是一个用 Verilog 描述的名为 modT 的数字系统框图。从图中可以看出模块 modT 含有在其他场合事先定义好的 3 个模块 modA、modB 和 modC，这里 modT 就是这 3 个模块所谓的上一级模块；3 个小模块有时候也被称为子模块。U1、U2 和 U3 代表每个模块在被模块 modT 调用时赋予的临时标记，标记名称是可以改变的，因为每一个模块可能会在其他场合被多次调用。

图 6.4 中每一个子模块的端口都处在上一级模块框图的内部，输入信号从左边或者底部进入系统，输出信号从系统右边或者顶部被引出。

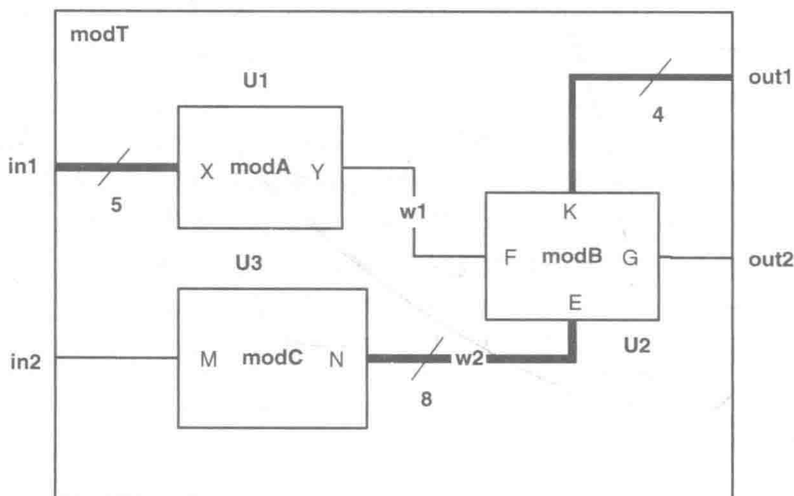


图 6.4 含有子模块的模块框图

代码 6.2 是针对图 6.4 的 Verilog 语言描述。

```
1 module modT (input [4: 0] in1,
2     input in2,
3     output [3: 0] out1,
4     output out2);
5 wire [7: 0] w2;
6 wire w1;
7 modAU1 (.X (in1), .Y (w1));
8 modBU2 (.F (w1), .E (w2), .K (out1), .G (out2));
9 modCU3 (.M (in2), .N (w2));
10 endmodule
```

代码 6.2 模块 modT 的 Verilog 语言描述

和之前的代码一样，每一行最左边的数字行标仅用作参考，它们不是模块源代码的一部分。前 4 行定义了模块 modT 的端口：in1 是一个 5 位的输入端，out1 是一

个 4 位的输出端，其他所有的端口都是 1 位的。第 5、6 两行定义了两个内部线网型变量，用于连接模块 modA、modB 和 modC。

模块的内部结构通过 7、8、9 三行的模块实例化语句来构建。每一行以被调用的模块名称开始，然后是空格，紧接着是临时标记名称（U1、U2、…）。

在 Verilog 里，主要有两种不同的方法来描述模块的连接关系：比较推荐的一种方法，被称为直接关联，就是代码 6.2 里所体现的。

具体方法是子模块通过“点”（.）和相应的信号进行直接关联，“点”的后面跟随的是端口名称，通过端口送往外部的信号在端口名称后面的括号里体现，相应的格式为：

```
Module - name instance - name (.port - name (net - name),
…);
```

直接关联相对于第二种方法（后面会提到）有两大主要优势：

- 所有连接均可以任意排序；
- 将端口名和对应连接的信号放在一起，减少了出错的可能性。

第二种定义模块的方法被称为位置关联。使用这种方法，每个模块的端口和模块对应的子模块之间的网络形成连接。举例说明，模块 modA 当使用位置关联时，表达方式如下：

```
modA U1 (in1, w1); //位置关联
```

很明显，从语体构架上看，位置关联没有直接关联语体健全，这是因为信号之间存在多种排列顺序的可能性。对于一些由于端口和信号的连接错误导致的位宽和端口方向的不完全匹配，Verilog 编译器并不会总是报错。

有时候，我们会将一些模块的端口悬空，这对于输入和输出端口都有可能。不管是用直接关联还是位置关联，悬空端口在端口声明里会用空格表示，其他信号的表达方法则和通常情况一样。下面给出两行带有悬空端口的模块定义，用两种关联方法来表达分别是：

```
//输出端口 K 是悬空的
modB U2 (.F(w1),.E(w2),.K(),.G(out2));
//输入端口 E 是悬空的
modB U4 (w1, , out1, out2);
```

当某个输入信号是悬空的情况下，Verilog 仿真器会将相应的信号设置为高阻 z。

正如之前介绍的，Verilog 使用两种主要的对象类型对硬件电路的信号进行建模：

线网型 wire（或 net）——附加在这类信号上的驱动必须是连续不断的。主要用途是建立连续赋值和模块之间的相互连接。

寄存器型 Reg——可以保留上一次的赋值。经常（但并不是专门）用在存储单

元上面。

对于模块端口的特性和层次化设计中对象的类型，在 Verilog 中都有一系列的规定。由于模块的内部限制，端口 input 和 inout 的默认类型是线网（wire）。模块的输出可以是线网型（wire），也可以是寄存器型（reg）。

模块的 output 和 inout 型端口必须和下一级的线网型连接。然而，输入端口可以用线网型（wire）和寄存器型（reg）两种类型的信号来驱动。

上述规则在图 6.5 中有详细说明。

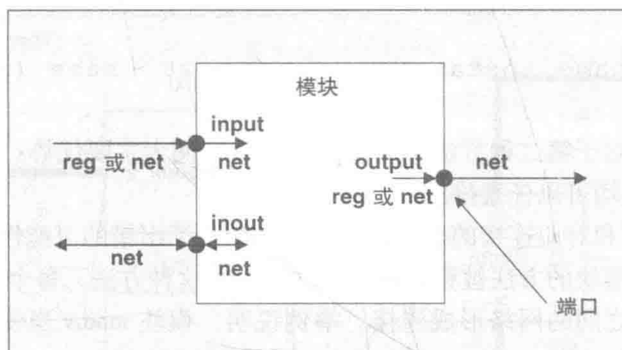


图 6.5 Verilog 端口连接属性规则

6.4 Verilog HDL 仿真：一个完整的设计过程

本节将向大家完整地介绍一个用 Verilog HDL 设计的案例，其中还包含测试固件的仿真。使用硬件描述语言（HDL）（例如 Verilog），的一大好处，就是它支持用户使用其强大的语言功能在设计过程中建立仿真环境。这就是所说的测试固件的概念（某些时候被称为测试平台或测试模块）。

测试固件的主要目的是为了验证整个设计的功能是否达到设计要求。通常可以简单地在输入端产生一个脉冲，来观察输出端的反应，或者用一些更加巧妙的方法来检测复杂系统中的微小错误。

测试固件最主要的好处在于，和设计时所用的语言一样，它也是用 Verilog 编写的，因此可以进行独立仿真，并有较大的灵活性，可以在任何支持 IEEE 标准的 Verilog 系统平台上运行。

图 6.6 是单比特二进制加法器的 Verilog 语言描述和模块框图[1]。模块 FA 用数据流形式来描述其中的逻辑行为。第 2、3 行的连续赋值语句分别用布尔代数的形式，来表达求和（sum）与进位（carry）等输出信号。这个模块是理想化的，因为信号的传输总是有延迟的。输入信号 A、B、Ci 上的任何变化都将触发 2、3 两

行赋值语句的执行，并在一个仿真时钟周期后更新输出信号 S 和 Cy 上的值。

```
1 module FA(output S, Cy, input A, B, Ci);

2 assign S = A^B^Ci;
3 assign Cy = (A&B)|(A&Ci)|(B&Ci);

4 endmodule
```

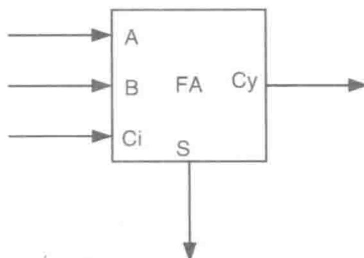


图 6.6 二进制全加器的 Verilog 代码和模块框图

图 6.6 所示的全加器也可以用其他方式实现，从低等级的 MOS 开关电路，到高等级的行为级描述均可以。因此，Verilog 对自顶向下的设计理念是完全支持的，这样的理念可以将概念化或者抽象的想法迅速地得到验证。随后设计方案经过优化，以更加具体的方式呈现，逐步向最终的硬件设计目的靠近。

学会设计单比特加法器之后，图 6.7 是将 4 个加法器串联起来，组成所谓的 4 位脉冲进位加法器[1]。

Add4 模块的端口（前两行）定义输入和输出的位宽都是 4 位（4bit），表示方式为 3：0。最低位的进位输入和最高位的进位输出是单比特信号。

4 位加法器是用 4 个全加器模块组成的，每个全加器模块的命名分别为 fa0、fa1、fa2 和 fa3。代码 4 ~ 11 行对这 4 个全加器进行了定义。它们通过进位向量 Cy（在第 3 行定义）进行内部连接，进位输入 Cin 和进位输出 Co 也是连续脉冲进位构架的一部分。

大家留意图 6.7 里用代码描述模块之间的互连所采用的直接关联和位选择的方法。例如，矢量输入 A 和 B 中的每一位都对应一个全加器的输入端，用代码来描述的话，就是在端口名称后面的括号里填写相应的矢量输入位，代表某一位数据的数字填写在方括号里：

`.A (Ain [1]) //FA` 模块里的端口 `A` 和输入矢量 `Ain` 的第二位相连。

尽管看起来比较烦琐，但是这样的方法避免了很多错误，显得比较清晰。

设计完成后，需要构建测试固件来验证系统的功能。表 6.3 和图 6.8 分别是测试代码和测试框图。

```
1    module Add4(output [3:0] Sum, output Co,
2           input [3:0] Ain, Bin, input Cin);

3    wire [2:0] Cy;

4    FA fa0(.S(Sum[0]), .Cy(Cy[0]), .A(Ain[0]),
5       .B(Bin[0]), .Ci(Cin));
6    FA fa1(.S(Sum[1]), .Cy(Cy[1]), .A(Ain[1]),
7       .B(Bin[1]), .Ci(Cy[0]));
8    FA fa2(.S(Sum[2]), .Cy(Cy[2]), .A(Ain[2]),
9       .B(Bin[2]), .Ci(Cy[1]));
10   FA fa3(.S(Sum[3]), .Cy(Co),    .A(Ain[3]),
11      .B(Bin[3]), .Ci(Cy[2]));

12   endmodule
```

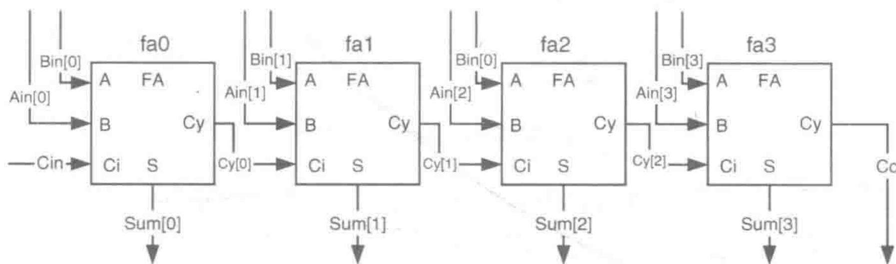


图 6.7 4 位加法器的 Verilog 代码和模块电路

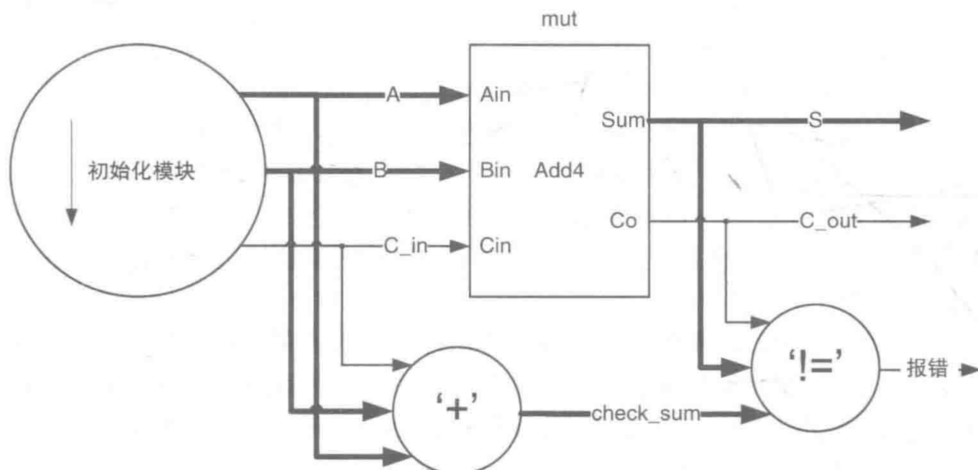


图 6.8 4 位加法器测试框图

```
1 'timescale 1ns/1ns
2 module Test_ Add4 (); // 测试模块不需要端口定义
```

```
3 //测试系统输入信号
4 reg [3: 0] A, B;
5 reg C_ in;

6//连接结果检测信号和报错信号
7 wire [4: 0] check_ sum;
8 wire error;
9 //输出响应
10 wire [3: 0] S;
11 wire C_ out;

12 integer test;

13 initial //只有在测试代码中出现，且只运行一次
14 begin
15    {A, B, C_ in} = 9'b000000000;
16    #100; //等待100个时间单位
17   for (test = 0; test < 512; test = test + 1)
18   begin //给所有的输入信号赋值
19      {A, B, C_ in} = test;
20      #100;
21   end
22    $stop;    //系统命令——停止仿真
23 end

24 //建立待测模块 (module - under - test)
25 Add4 mut (. Sum (s), . Co (C_ out), . Ain (A), . Bin (B),
26.       Cin (C_ in));

27 //将输入信号用内部操作符‘+’相加
28 assign check_ sum = A + B + C_ in;
29 //和待测模块的输出做比较
30 assign error = (check_ sum ! = {C_ out, S});
31
32 endmodule
```
代码6.3 4位加法器Verilog测试代码

图 6.8 是系统的测试框图。原有的模块名称在测试固件里会被称为待测模块（module - under - test）或直接缩写为 mut，因此图中的 Add4 模块上方被标记为 mut。测试代码会产生一组输入脉冲，对 A、B 和 Cin 进行连续赋值，让输入信号产生变化，这种模拟行为的代码被称之为初始化时序模块，图中最左边（含有 initial block）的圆圈则代表了这一部分代码。

为了验证 4 位加法器的操作是正确的，内嵌的 Verilog + 操作符被用作求和计算，得出的 5 位结果以 check_ sum 来命名。

Check_ sum 随后被用作与 4 位加法器的输出进行比对，这里用到的比较操作符为！ =（不等于）。当两个比较对象不相等的情况下，输出信号 error 会给出提示。这样，构建的简易测试固件可以验证 Add4 模块的工作是否正常。

代码 6.3 中涉及的一些 Verilog 的语法结构，会在后面向大家进一步阐述。更多的关于 Verilog 语言的特性，将在第 7 和第 8 章详细介绍。

测试代码的第一行是所谓的编译指令。这样的特殊指令和 C/C + + 语言中的预处理指令的目的类似。然而，找不到熟悉的“#”标记，Verilog 使用重音符“`”来表示这样的指令。timescale 定义了时间量程和时间精度，精度在量程后面，用“/”隔开。这里两个单位是相同的，都是纳秒（ns）；这意味着模块内产生的任何时延在仿真器看来都为 1ns。时间精度最小可以设定为毫微微秒（10^{-15}s），因此可以进行精度非常高的仿真。不过这里的加法器不需要这么高的精度。

由于没有定义输入和输出端口，因此第二行代码仅声明了 Test_ Add4 属于测试代码，而不是普通的设计代码。模块名后跟随的括号不是必需的，因此如果没有括号不会产生语法报错；而分号在一行代码的结尾是必需的。

代码第 4、5 行声明了加法器模块的输入脉冲信号。寄存器型（reg）表明了这些信号在其赋值被更新之前必须保持原来的值，赋值语句是从代码第 13 行的初始化（initial）开始的。

待测模块的输出是用连续赋值语句来驱动的（第 25、26、28、30 行），因此这些信号必须被定义为线网型（wire）（第 7、8、10、11 行）。

测试固件的代码主体是第 13 到 23 行。这部分代码中所有信号都是寄存器型（reg），这样它们在仿真运行的过程中会一直保持上一次的赋值，方便用户观察测试结果。

initial 引领的代码模块在仿真过程中按语句排列的顺序执行，且只执行一次，这意味着这类代码只能用作测试代码；它们无法直接用硬件电路来体现。代码模块从第 15 行开始执行，此时仿真计时的起点是 0ns。所有输入信号的初始化均通过下面一行语句完成：`{A, B, C_ in} = 9'b000000000`。

将 3 个信号合并统一赋值为逻辑 0，是利用组合操作符 {} 来实现的。

代码第 16 行代表仿真在运行的过程中在这个位置做短暂停留，延迟时间为 100ns；目的是让待测模块对输入信号的变化做出回应（产生结果）。井号（#）

表示延迟的特殊符号。

在延迟之后，代码第 17 到 21 行是一个 for 的循环，循环变量为整形（integer）变量 test，它从 0 递增到 511。整形变量（integer）是一个系统默认的字节变量，表示一个整数（通常的长度是 32 位），特性和寄存器型（reg）类似，也可以在运行过程中暂时保留原先的值。

```
for (test = 0; test < 512; test = test + 1)
begin //给所有输入信号赋值
   {A, B, C_ in} = test;
  #100;
end
```

for 循环的主体用 begin 和 end 来封装。代码第 19 行将整形变量 test 的低 9 位赋值给输入信号，这种复合型赋值方式在 Verilog 里是允许的。循环里的第二行语句的作用是在主体进入下一个循环之前延迟 100ns。这样，加法器的输入端赋值从二进制“000000000”到二进制“111111111”（十进制 511）循环递增，且每一次赋值后延迟的时间都是 100ns。

变量 test 的值在循环的末尾递增，在循环的开始测试是否符合 for 的循环条件；因此当 test 的值达到十进制 512 时，条件语句 test < 512 不再成立，循环结束。一个比较重要的问题是 for 语句的无限循环，即永远循环下去。这种情况会在 test 是一个长度只有 9 位的寄存器型（reg）变量而不是 32 位整形（integer）变量时发生。因为循环里 test 变量的位宽和输入信号的组合位宽都是 9 位，因此这里 test 如果只有 9 位逻辑上也是说得通的。

然而，当 test 的值递增到二进制“111111111”时会出现问题。

此时循环还会继续下去，因为再递增一次会将 test 的值变为二进制“000000000”，作为一个 9 位无符号二进制数，它的值溢出了。而循环的终止条件 test < 512 将永远不会成立，此时变量的值永远不会超过十进制 511。所以，如果是一个 9 位的寄存器型（reg）变量在循环里持续运行，而不是一个整形（integer）变量，仿真器将永无休止地循环下去，并且占用大量磁盘空间来存储仿真结果。

一种可能的解决方案是将 test 定义为 10 位寄存器型变量，多出来的一位可以让变量达到终止循环的条件“1000000000”。

当循环足够多的次数之后，仿真会自动停下来，通过一条系统任务语句来实现（第 22 行）

```
$stop;
```

系统任务的标记和美元（$）的符号类似，且有广泛的用途，可以设定时序（例如 $setup（），$hold（）等），也可以将仿真结果导出到目标文件（例如 $dumpvars，$dumpfile（“filename”））。这里的 $stop 通常在测试固件里被用来强制终止仿真；其他类型的系统任务将在第 7 和第 8 章有更多的展现。

仿真最终结果的验证在代码的第 28 行和第 30 行体现：

```
assign check_ sum = A + B + C_ in;
assign error = (check_ sum ! = {C_ out, AS});
```

上述两条赋值语句产生一个诊断信号 error，当4 位加法器产生的结果和系统内嵌的加法运算结果不一致时，信号 error 会拉高报错。基于本系统较为简单，尽管看起来可能必要性不大，但也从侧面体现出 Verilog 语言对于复杂系统设计验证的强大功能。

4 位加法器模块及其测试固件需要和 Verilog - 2001[9] 版本兼容的仿真工具。其他开发商提供的仿真软件也可以用。表 6. 1 列出了几个比较常用的仿真工具。

表 6. 1 比较常用的几种 Verilog 仿真工具

名称	开发商	网址
active - HDL® student edition	Aldec Incorporated	http：//www. aldec. com/education/students/
Modelsim - PE student②®	Mentor Graphics	http：//www. model. com/resources/student_ edition/download. asp
Verilogger®	Synapticad	http：//www. syncad. com/syn_ down. htm
Xilinx® ISE Simulator①	Xilinx	http：//www. xilinx. com/ise/logic_ design_ prod/webpack. htm

① ISE 仿真器是 Xilinx® 官方网络版可编程逻辑设计安装包里的一部分，因此必须完整安装网络版的设计软件才能使用。

② 也可以使用 Modelsim® 专门为 Xilinx® 定制的开发商专用版（ModelsimXE®）。

不管用哪种工具，在仿真之前都必须建立 Verilog 源文件。大家必须要养成良好的习惯，将编写完成的代码源文件以 ASCII 文本文件的格式存储（一般命名方式为“module - name. v”）。大部分仿真工具的文本界面支持自动上下文相关编辑，例如将关键字用彩色字体高亮，自动加载行数，自动缩进以及编排语言格式等。所有这些辅助功能都将帮助设计者更好地理解和维护复杂的系统设计。

大部分仿真器在仿真之间都要事先建立一个工程。建立工程就是将所有设计所用到的文件集合在一起，放在同一个文件夹里。源文件编写完成后，可以添加到工程目录下，随后就是编译。编译的步骤和其他高级语言开发系统相似：目的是建立一个可执行文件，方便加载到仿真内核里，当然前提是不出现任何语法和结构的错误。

所有 Verilog 仿真器都能够以波形图来输出仿真结果。图 6. 9 就是 4 位加法器的仿真结果波形图。

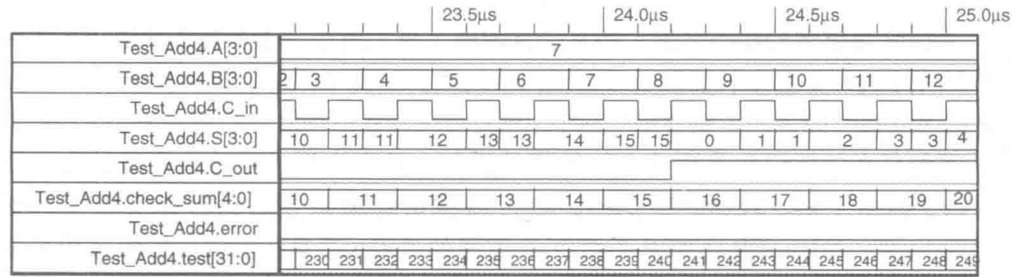


图 6. 9 4 位加法器部分仿真结果示意图

参 考 文 献

1. Wakerly J.F. Digital Design: Principles and Practices, 4th edition. New Jersey: Pearson Education, 2006.
2. www.systemc.org [2007 October].
3. www.systemverilog.org [2007 October].
4. www.verilog.com [2007 October]. (Links to IEEE Standards site and other Verilog information.)
5. www.accellera.org/home [2007 October].
6. www.adacore.com/home/ada_answers/ada_overview [2007 October].
7. www.synopsys.com/products/logic/design_compiler.html [2007 October].
8. Weste N.H.E., Eshraghian K. Principles of CMOS VLSI Design. Addison Wesley, 1993; Section 6.2.
9. Ciletti M.D. Advanced Digital Design with the Verilog HDL. New Jersey: Pearson Education, 2003; Appendix I – Verilog-2001.

除了上述参考文献，网络上还有大量具有参考价值的关于 Verilog 的教材和链接，其中大部分可以通过在搜索网站里输入关键字“Verilog”来找到。

第7章　Verilog HDL 体系

本章向大家介绍 Verilog HDL 的一些基本特性。和其他高级语言一样，Verilog 定义了一系列类、操作符和组织构架等形成整个语言体系。因此本书将重点放在描述组合逻辑和时序逻辑上面。

7.1　内置基本单元和类

7.1.1　Verilog 的类

上一章提到，Verilog 两种最基本的类是线网型和寄存器型。通常来说，线网型用来建立连接，而寄存器型用来存储信息，尽管寄存器型并不总是代表着时序逻辑。

每一种类自身又可以衍生出不同特性的类。表 7.1 列出了所有类的名称，这些名称在 Verilog 语系里已经被固化，其中最常用的类用粗体字标明。

表 7.1　Verilog 里的类型

线网型（建立连接）	寄存器型（存储数据）
wire	
tri	
supply0	reg
supply1	integer
wand	real
wor	time
tri0	realtime
tri1	
triand	
trireg	
trior	

表示内部连接的类除了线网型 wire，还有两种用来定义电源：supply0 和 supply1。

这种特殊的线网型变量驱动能力很强（意味着它们的值无法被其他类型的信号改写），用所谓的“电源（supply）”来命名，主要作用是将输入端的信号上拉或下拉。下面一小段程序告诉大家在 Verilog 中如何使用电源网络：

```
module …

supply0 gnd;
supply1 vdd;
  nand g1 (y, a, b, vdd); //将与非门的其中一个输入拉高

endmodule
```

电源网络还可以在使用Verilog描述开关级MOS电路时发挥作用。不过大部分综合设计的工具不支持Verilog里自带的开关单元[1]（例如nmos，pmos，cmos等），因此这些类型不在本书的讨论范围内。

在表7.1左边，除了已经介绍的线网类型外，大部分是一些比较高级的建模类型，综合工具都不支持。不过也有例外，就是其中的tri类型。这个类和基本的wire型类似，一般使用时会让语言描述更加清晰明了。这两种线网类型支持多种驱动形式（连续赋值语句、基本单元和单个模块等），当没有外部驱动给予一个明确的逻辑值时，它们将维持高阻。tri类型可以替代wire类来表示网络长期处于高阻的状态。

类似于wire和tri这样的类型一般不支持初始化赋值；在仿真初期，这些网络的初始值都是高阻。

对于多路驱动和高阻状态的处理已经成为了Verilog HDL体系的一部分，而不像其他硬件描述语言，需要IEEE标准支持的额外插件来定义和处理上述功能。

表7.1右侧主要是Verilog里的寄存器类型。它们通过时序语句赋值，并能够保持上一次的赋值结果，因此专门用在时序模块里。最常用的两个寄存器类型就是reg和integer，其他的由于综合工具不支持，因此不在讨论范围内。

reg和integer之间的一些明显的区别是：让reg型变量在许多场合下更加受到青睐。

一个reg型变量可以定义成单比特（即当没有特别指定位宽的情况下）或者矢量对象，请看下面的语句：

```
reg a, b; //单比特寄存器变量
reg [7: 0] busa; //一个8位寄存器变量
```

可以看出，一个reg变量可以是任意的位宽，而且不受处理器的字宽限制。

而一个integer变量，正常情况下不能指定位宽，通常它和处理器的默认字宽一致，要么为32位，要么为64位。

它们其他的区别体现在运算表达式里的处理方式。integer变量通常是以带符号的2的补码形式存储，参与运算时也同样如此，即以一个带符号的数出现（但表达式里其他变量也必须是带符号的）。相反，reg型变量通常默认是无符号的。

如果用reg型或者wire型变量参与带符号（signed）的2的补码的运算，通常

需要在声明变量类型时体现出它们的正负（带符号）。这样既可以进行带符号的运算，也不用担心 integer 变量所带有的处理器位宽限制，例如：

```
reg signed [63: 0] sig1; //一个 64 位带符号寄存器变量
wire signed [15: 0] sig2; //一个 16 位带符号线网变量
…
```

这里关键词 signed 代表变量可以是正的，也可以是负的。对于模块端口的定义也适用，例如：

```
module mod1 (output reg signed [11: 0] dataout,
       input signed [7: 0] datain,
       output signed [31: 0] dataout2);
…
```

最后还要说明的是，integer 和 reg 都可以被初始化赋值，而作为 reg 型，初始化赋值可以出现在模块的端口声明中，例如：

```
module mod1 (output reg clock = 0,
       input [7: 0] datain = 8'hff,
       output [31: 0] dataout2 = 0);
integer i = 3;
…
```

因此在 Verilog 里，reg 和 integer 所适用的场合有一定的区别。通常来说，reg 变量用来构建实际存在的硬件寄存器，比如计数器、状态寄存器和数据链路寄存器等。而 integer 更多地用在描述算术运算方面，比如循环计数。代码 7.1 向大家展示了这两种变量的基本用法。

这里的 Verilog 代码描述了一个 16 位同步二进制计数器。内部使用了两个 always时序模块——本书会在第 8 章详细阐述时序模块的语言描述方法。

第一个时序模块从第 5 行到第 11 行，描述了一组被“clock”输入信号的上升沿（从逻辑 0 变为逻辑 1）驱动的触发器。

这些触发器的状态全部被存入一个 16 位 reg 型输出变量 q 里，q 的定义在程序的第一行。另一个 16 位 reg 型变量 t 的定义在第三行，它是第二个时序模块所构建的组合逻辑电路的输出（第 12 行到第 20 行）。这里阐述的一个观点是 reg 型变量通常在时序模块里充当保持上一次赋值的角色时，并不总是代表时序逻辑。从第 12 行开始的 always 模块对输出信号 q 的变化产生响应，每次 q 的值出现更新，则的值也产生变化。而更新后的 t 值会决定下一个时钟上升沿到来时，q 的值如何变化（对应第一个 always 模块）。

```
1 module longcnt (input clock, reset, output reg [15: 0 [ q];
2
3 reg [15: 0] t; //触发器输入和输出
```

```
4 //时序逻辑
5 always @ (posedge clock)
6 begin
7    if (reset)
8        q < = 16'b0;
9    else
10       q < = q ^ t;
11 end

12 always @ (q) //组合逻辑
13 begin : t_ block
14        integer i;   //整型变量用作循环计数器
15        for (i = 0; i < 16; i = i + 1)
16          if (i = = 0)
17           t [i] = 1'b1;
18        else
19           t [i] = q [i -1] & t [i -1];
20 end
21 endmodule
```

代码 7.1 reg 和 integer 型变量在 Verilog 里的应用

第二个时序模块（第 12 行到第 20 行）被称之为被命名的模块，大家可以看到第 13 行 begin 后面多出了冒号，随后的 t_ block 就是模块名。通过这种命名方式，局部定义的寄存器型变量（reg）和整型变量（integer）被限定只能在模块内部使用。这里，整型变量 i 作为 for 循环的变量，从第 15 行到第 19 行，参与了寄存器变量 t 除最低位 t［0］以外的每一位赋值运算。循环中，变量 t 的第 i 位赋值是 q 的第［i－1］位和 t 的第［i－1］位相与的结果。因此整个 always 循环描述的是一个迭代的同步二进制计数器。

图 7.1 是根据代码 7.1 绘制的计数器系统框图。

在此基础上形成的 4 位计数器仿真结果，如图 7.2 所示。图中很清晰地显示了输出 q 以二进制的格式计数叠加，以及矢量 t 产生的相应脉冲，在适当的时刻触发输出信号 q 的数据位进行状态转换。例如，当 q 的输出为二进制 0111 时，t 的值为二进制 1111，当下一个时钟的上升沿到来时，输出 q 的每一位都将切换状态。

7.1.2 Verilog 逻辑值和数字值

Verilog HDL 里 reg 型或者 wire 型变量中的每一位数据的值都可能是表 7.2 里 4 种中的一种。Verilog 同时还提供了内置的信号强度建模，但这种建模对综合工具

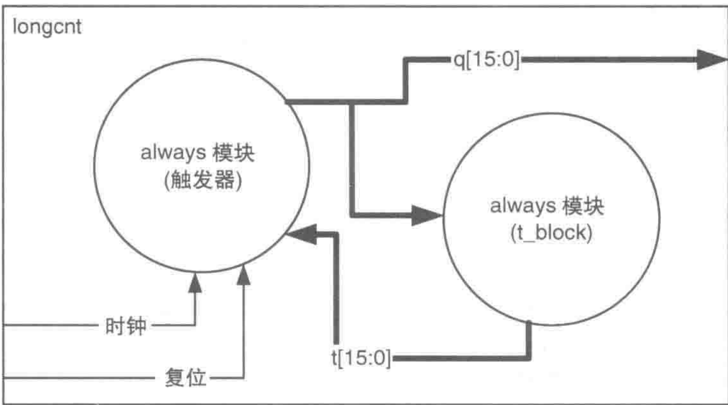


图 7.1　模块 longcnt 的系统框图

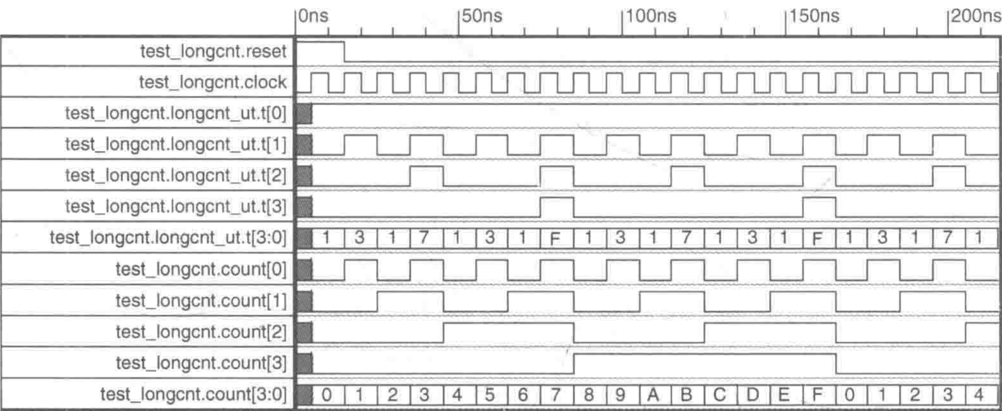


图 7.2　模块 longcnt（4 位版本）的仿真结果

来说应用范围不广，因此本书不做讨论。

表 7.2 里，逻辑 0 和逻辑 1 分别代表了布尔代数中的“低”和“高”。事实上，任何非零的值在 Verilog 里都可以被认作有效的，这和 C/C++ 语言里的规定是一样的。逻辑关系操作符运算得出的单比特结果，决定最终是高（1）还是低（0）。

除逻辑 0 和 1 以外还定义了两个类型的逻辑值，它们用来模拟未知状态（x）和高阻（z）。其中 x 在某些场合也表示自由态的意思，其含义是指可以任意赋值。在仿真的开始，即 0 时刻，所有寄存器被复位或者初始化到未知状态，除非它们在模块定义初期就被明确赋值。从另一个角度来说，线网型变量的初始化状态总是高阻态 z。

表 7.2　4 种既定逻辑值

逻辑值	含义
0	逻辑 0 或者低
1	逻辑 1 或者高
x	未知（或自由态）
z	高阻

一旦仿真开始，所有寄存器型和线网型信号在正常情况下，都会被赋予有效的数值或者高阻 z，而如果仿真过程中出现未知态 x，则通常预示着设计内部行为或者结构出现问题。

未知态 x 偶尔也会被故意地赋给某个信号，作为模块内部的一种定义方式。这种情况下，x 则代表了所谓的自由态，主要目的是在综合电路的过程中方便编译流程，简化电路。

Verilog 同时还提供了一组内置的预设逻辑门单元，这些基本单元对于未知态和高阻态的反应都在合理的预估范围内。图 7.3 给出了一个非常简单的二输入与门模块，使用的是内置的与门单元来构建。图中仿真结果清晰地显示了模块针对输入信号为 x 和 z 时的不同响应。

```
1 module valuedemo (output y , input a, b);
2 and g1 (y, a, b);
3 endmodule
```

根据图 7.3，在 0 ~ 250ns 的时段，与门输出 y 的结果和对应输入 a 和 b 是匹配的。在 250ns 的时刻，输入 a 被赋值高阻 z（图中虚线表示），而此时的输出在 300 ~ 350ns 区间内为 x（阴影部分），因为逻辑 1 和 z 经过与门的结果是未知的。同样地，在 400 ~ 450ns 区间内，输入 b 的值为 x 导致输出 y 也为 x。

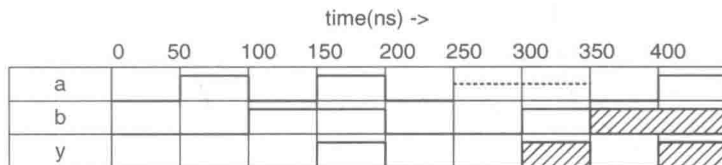


图 7.3　高阻态 z 和未知态 x 经过与门的仿真结果

然而，在 350 ~ 400ns 和 250 ~ 300ns 两个时段，由于其中一个输入信号为低，所以导致输出 y 为低。原因大家很清楚，与门有逻辑 0 参与的情况下输出结果都是逻辑 0。

7.1.3 如何赋值

在Verilog HDL里有两种类型的数字：带位宽和不带位宽。那么一个带有位宽的数字格式定义为：

<size>'<base><number>

<size>和<base>两个选项不是必需的，如果没有特别规定，这两个参数缺失的情况下，后面的数字表示十进制，且变量的位数和赋值的位数是统一的。

<size>本身是一个十进制数，表示数字基于二进制的长度。<base>有4种定义，具体如下：二进制为b或者B；十六进制为h或者H；十进制（默认）为d或者D；八进制为o或者O。

变量实际的数值是用（0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F）中的某些组合来表示的。十六进制数涵盖了所有的选项，而二进制、八进制和十进制数分别被限制只能用（0、1），（0~7）和（0~9）等部分组合。下面给出一些完整的赋值表达方式：

```
4'b0101 //4位二进制数
12'hefd //12位十六进制数
16'd245 //16位十进制数
1'b0, 1'b1 //逻辑0和逻辑1
```

通常来说，如果变量被定义为整型（integer），我们不需要特别规定它是多少位，因为这类变量是没有位宽的（通常是32位或者64位，取决于处理器平台）。

对于x和z（未知态和高阻态），通常表达它们的格式是二进制、十六进制和八进制。如果是二进制，则只有1位，八进制对应3位，而十六进制对应4位。

此外，如果变量的最高位是0、z或者x，则变量会自动在高位补齐，这样高位部分的每一位都是一样的。

下面给出一些带有x和z的赋值实例：

```
12'h13x //12位十六进制数，用二进制表示为'00010011xxxx'
8'hx //8位十六进制数，用二进制表示为'xxxxxxxx'
16'bz //16位二进制数'zzzzzzzzzzzzzzzz'
11'b0 //11位二进制数'00000000000'
```

单比特二进制逻辑0和逻辑1的表示方法通常为：

```
1'b0 //逻辑0
1'b1 //逻辑1
```

当然十进制的0和1也可以用上述方法表示。正如之前提到的，在Verilog语言里，1'b0和1'b1对应了布尔代数里的“低”和“高”。

7.1.4 Verilog HDL 基本门电路

Verilog HDL 内部提供了一系列整套的基本逻辑单元和三态门，用来描述数字门电路。这些基本单元都是可以综合的，然而它们更多地出现在综合工具输出的门级 Verilog 网表里。

图 7.4 和图 7.5 列出了每个基本单元的符号和 Verilog 语言格式[1]。图中对于基本逻辑单元的使用方面非常直观。最基本的逻辑门，例如与门、或门等，都是单路输出，但是允许多路输入（图 7.4 只有两路输入）。缓存器和非门允许多路输出、单路输入。

三态门都是 3 个端口：输出、输入和控制。控制端口的作用是决定三态门的输出是高阻态还是正常输出。

对于所有基本的逻辑门单元，输出端口必须是线网型，通常是 wire 型，但是输入端口可以是连线网型也可以是寄存器型。

在逻辑门单元和模块之间可以加入一些延迟，延迟的形式可以是单个延迟，也可以针对不同的情况的不同延迟，例如上升时段延迟、下降时段延迟、开关延迟[1]等。下面举个例子来说明这样的情况：

```
//与门的输出上升时段延迟 10 个时间单位
//且下降时段延迟 20 个时间单位
and # (10, 20) g1 (t3, t1, a);
//三态缓存器的输出上升时段延迟 15 个时间单位
//下降时段延迟 25 个时间单位
//开关时段延迟 20 个时间单位
bufif1 # (15, 25, 20) b1 (dout, din, c1);
```

图 7.6 是一个利用内置基本逻辑单元以及 Verilog 语言来描述的简易逻辑门电路。代码从第 4 行到第 8 行，每一行都描述了一个基本逻辑门电路。每一个逻辑门都带有 10ns 的传输延迟，这意味着输出的值会在输入信号变化 10ns 后得到更新，不管输出信号是上升还是下降。在仿真过程中实际使用的时间单位，是在测试平台用 timescale 编译向导来规定的，这对于模块来说是立刻生效的（具体参考代码第 1 行）。

逻辑门符号	Verilog 实例
in1, in2, out	**and** a1(out, in1, in2);
in1, in2, out	**nand** na1(out, in1, in2);
in1, in2, out	**or** o1(out, in1, in2);
in1, in2, out	**nor** no1(out, in1, in2);
in1, in2, out	**xor** xo1(out, in1, in2);
in1, in2, out	**xnor** xno1(out, in1, in2);
in, out1	**not** nt1(out1, in);
in, out1	**buf** b1(out1, in);

图 7.4　Verilog 基本逻辑门

逻辑门符号	Verilog 实例
in, out, ctrl	**bufif1** g1(out, in, ctrl);
in, out, ctrl	**bufif0** g1(out, in, ctrl);
in, out, ctrl	**notif1** g1(out, in, ctrl);
in, out, ctrl	**notif0** g1(out, in, ctrl);

图 7.5　Verilog 基本三态门

```
1    `timescale  1 ns /1 ns

2    module x_or_s(output y, input a, b);

3    wire t1, t2, t3, t4;

4    and #10 g1(t3, t1, a);
5    and #10 g2(t4, t2, b);
6    not #10 g3(t1, b);
7    not #10 g4(t2, a);
8    or  #10 g5(y, t3, t4);

9    endmodule
```

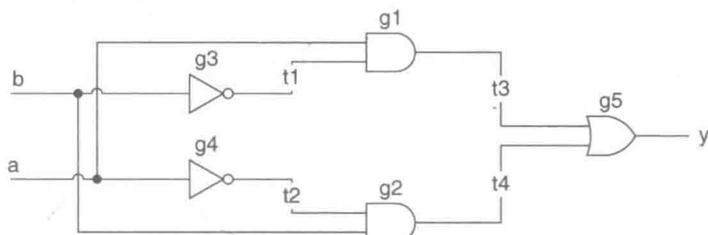


图 7.6　逻辑门电路和 Verilog 语言描述

逻辑门延迟也是有惯性的，意思是输入端脉冲的脉宽如果小于或者等于逻辑门延迟的时长，将不会影响输出的结果，即输入端的变化不会改变输出结果。这种情况也符合实际逻辑门的物理特性。

像图 7.6 里的门级延迟由于被固化，因此比较直观，通常用来检验逻辑门电路的性能，例如一些 TTL 分立逻辑器件。然而，Verilog HDL 作为逻辑综合工具软件的输入载体，其本身在通常情况下是不带有任何传输延迟值的，因此这些传输延迟一般会被综合工具忽略。

7.2　操作符和描述语句

Verilog HDL 在建立硬件模块方面提供了一系列功能强大的操作符。表 7.3 列出了完整的 Verilog 操作符分类。表格分 4 列，从左到右分别是操作符类型、操作符号、操作符描述和适用的操作数个数。

通过表 7.3，可以发现这些操作符和 C/C + + 语言里使用的非常类似。但也有一两个比较突出的区别和改进。以 C 为基础的语言和 Verilog 在操作符上的主要区别，在表 7.3 之后有详细总结。

表 7.3　Verilog 操作符

<table>
<tr><th>操作符类型</th><th>符号</th><th>具体操作</th><th>操作数</th></tr>
<tr><td>运算型</td><td>*</td><td>乘</td><td>2</td></tr>
<tr><td></td><td>/</td><td>除</td><td>2</td></tr>
<tr><td></td><td>+</td><td>加</td><td>2</td></tr>
<tr><td></td><td>-</td><td>减</td><td>2</td></tr>
<tr><td></td><td>%</td><td>取模</td><td>2</td></tr>
<tr><td></td><td>* *</td><td>幂指数</td><td>2</td></tr>
<tr><td>逻辑型</td><td>!</td><td>逻辑非</td><td>1</td></tr>
<tr><td></td><td>&&</td><td>逻辑与</td><td>2</td></tr>
<tr><td></td><td>| |</td><td>逻辑或</td><td>2</td></tr>
<tr><td>关系型</td><td>></td><td>大于</td><td>2</td></tr>
<tr><td></td><td><</td><td>小于</td><td>2</td></tr>
<tr><td></td><td>> =</td><td>大于等于</td><td>2</td></tr>
<tr><td></td><td>< =</td><td>小于等于</td><td>2</td></tr>
<tr><td>对等型</td><td>= =</td><td>等于</td><td>2</td></tr>
<tr><td></td><td>! =</td><td>不等于</td><td>2</td></tr>
<tr><td></td><td>= = =</td><td>全等于</td><td>2</td></tr>
<tr><td></td><td>! = =</td><td>不全等于</td><td>2</td></tr>
<tr><td>位操作</td><td>~</td><td>按位逻辑非</td><td>1</td></tr>
<tr><td></td><td>&</td><td>按位逻辑与</td><td>2</td></tr>
<tr><td></td><td>|</td><td>按位逻辑或</td><td>2</td></tr>
<tr><td></td><td>^</td><td>按位异或</td><td>2</td></tr>
<tr><td></td><td>^~或~^</td><td>按位同或</td><td>2</td></tr>
<tr><td>递减型</td><td>&</td><td>与</td><td>1</td></tr>
<tr><td></td><td>~&</td><td>与非</td><td>1</td></tr>
<tr><td></td><td>|</td><td>或</td><td>1</td></tr>
<tr><td></td><td>~|</td><td>或非</td><td>1</td></tr>
<tr><td></td><td>^</td><td>异或</td><td>1</td></tr>
<tr><td></td><td>^~或~^</td><td>同或</td><td>1</td></tr>
<tr><td>位移型</td><td>> ></td><td>右移</td><td>2</td></tr>
<tr><td></td><td>< <</td><td>左移</td><td>2</td></tr>
<tr><td></td><td>> > ></td><td>带符号右移</td><td>2</td></tr>
<tr><td></td><td>< < <</td><td>带符号左移</td><td>2</td></tr>
<tr><td>组合型</td><td>{}</td><td>组合</td><td>任意值</td></tr>
<tr><td>复制型</td><td>{{}}</td><td>复制</td><td>任意值</td></tr>
<tr><td>条件型</td><td>?:</td><td>条件判断</td><td>3</td></tr>
</table>

• Verilog 提供了一些功能强大的一元逻辑操作符（所谓简化操作符），这些操作符是对同一个字里每一位数据进行相应的逻辑运算操作。

• 外加的全等于/不全等于操作符是为了处理高阻态（z）和未知态（x）而特别设立的。

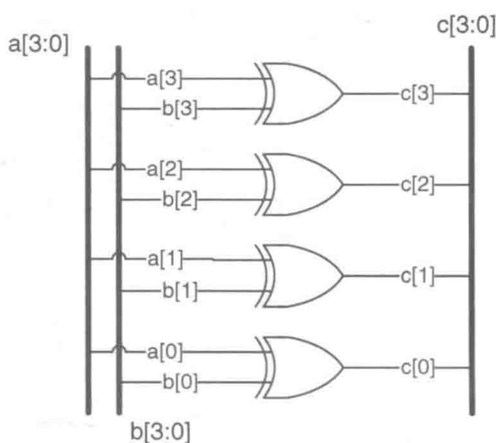
• 大括号 {} 是用于组合和复制的操作符，取代了（一般用 begin…end 来引导）模块化语句。

表 7.3 里的操作符和对应的操作数组成的语句，一般出现在连续赋值语句的右边或时序语句内部。

参与运算的操作数可以是线网型（wire）、寄存器型（reg）和整型（integer）的任意组合，但是语句所描述的对象必须是线网型（wire）和寄存器型（reg）其中的一种，且取决于代码用的是连续赋值语句还是时序语句。

对于含多位数据的情况（例如总线），操作数可以是整个对象（此时操作对象就是对象名称），也可以是其中的一部分数据位（多路总线对象的其中一部分）或者其中的某一位，下面举例来说明。

针对下面的变量声明，图 7.7 ~ 图 7.10 给出一些连续赋值语句和其对应的逻辑电路



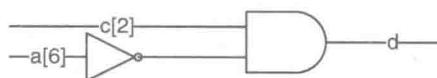

```
// 4位2输入异或门
assign c = a[3:0] ^b[3:0];
```

图 7.7 部分变量参与计算的异或门

```
wire [7:0] a, b;    //两个 8 位线网型变量
wire [3:0] c;    //4 位线网型变量
wire d;    //单比特线网型变量
```

图 7.7 是对两个 4 位操作数按位进行异或操作。根据电路图，a 和 b 分别是 8 位线网型变量，参与计算的是各自的[3:0]部分，输出 c 的每一位都是由相应的 a 和 b 的每一位按位异或计算得出。

线网型（wire）和寄存器型（reg）变量里某一位数据的访问是通过位选操作符（方括号 [ ]）来实现的。图 7.8 是用连续赋值语句来描述一个带有非门输入端的与门。




```
//2输入与门，其中一个输入取反
assign d = c[2] & ~ a[6];
```

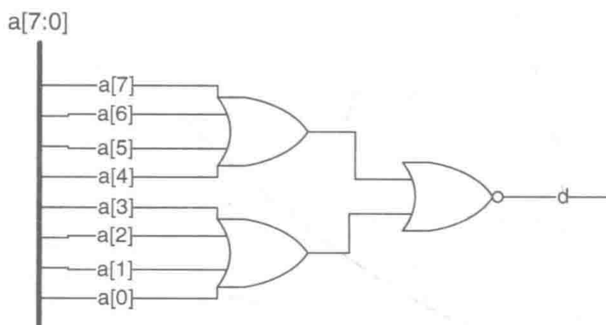
图 7.8 位选择参与计算的与门

表 7.3 中的按位递减操作符是 Verilog HDL 里独有的。它们提供了一种处理变量

中每一位数据的办法。图 7.9 是递减或非运算的例子。通过将操作数里每一位进行或运算，最终结果是一位数据，随后将此结果取反，等效语句如下：

```
assign d = ~(a[7] |a[6] |a[5] |a[4] |a[3] |a[2] |a[1] |a[0]);
```

图 7.9 中如果输入 a 的每一位都为逻辑 0，输出信号 d 的最终结果为逻辑 1，否则 d 的结果为逻辑 0。所有递减操作都位于赋值语句操作符的右边。带有取反操作的，例如或非递减（~ |）、与非递减（~&）等，递减操作优先于取反操作执行。对于多位逻辑运算，按位递减提供了一种比较方便的处理方法，用户不需要重新定义新的逻辑门单元。还要说明的是，如果操作数中任意一位或几位的值为高阻（z）状态，系统会默认那部分数据的状态为未知（x）。




```
//8输入或非门，使用按位或非方法输出
assign d = ~|a;
```

图 7.9 递减或非操作

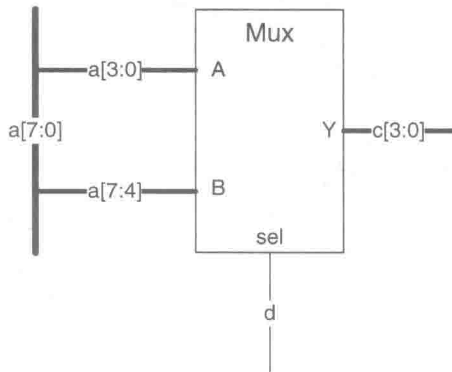
图 7.10 里使用了条件操作符（?:）来描述 4 个 2 选 1 多路选择器，当控制信号（这里是信号 d），为逻辑 1（1'b1）时，数据的低四位 a[3:0]会被赋值给 c。当信号 d 的值为逻辑 0（1'b0），输出 c 的赋值为高四位 a[7:4]。

不含符号的位移操作符（表 7.3 中的 > >，< <）将多位线网型变量和寄存器型变量进行位移，具体移动多少位由操作符后面的第二个操作数来决定。移位过程中用逻辑 0 来填补空缺，因此在面对 2 的补码（带符号）时必须格外注意。如果一个数为负，且为 2 的补码，右移操作后，代表符号的那一位会从 0 变成 1，其值会从负数变为正数。

2 的补码在右移时可以用带符号的操作符来完成‘ > > >’（其中线网型和寄存器型必须事先声明是带符号的），或者还有一种方法就是用复制/连接操作符（后面会介绍）。

下面的语句描述的是如何运用没有符号的移位操作：

```
//声明和初始化变量 X
reg [3: 0] X = 4'b1100;
```




```
// 8位输入4位输出多路选择器，并带有高低位选择信号
assign c = d ? a[3:0] : a[7:4];
```

图 7.10 8 路输入 4 路输出的多路选择器

```
Y = X > > 1; //结果是 4'b0110
Y = X < < 1; //结果是 4'b1000
Y = X > > 2; //结果是 4'b0011
```

Verilog 支持用在多位寄存器型变量（reg）和线网型变量（wire）的算术运算，同时也包括整型（integer）变量；这使得在使用 Verilog 描述例如计数器（+，-和%）和数字信号处理（*和/）等硬件系统时变得非常有用。

除了整型（integer）以外，所有运算操作符和比较操作符默认这些常用变量类型都是无符号的。不过，正如之前 7.1 节里提到的，寄存器型和线网型（模块端口也适用）可以带有符号。通常说来，只有当语句里所有的变量都是有符号的，Verilog 才会进行带符号的运算。如果其中一个操作数是无符号的，Verilog 会用系统函数 $signed（）根据需要进行转换（另一种系统函数$unsigned（）的功能与其相反）。

代码 7.2 是运用乘法、除法和移位等操作符对有符号和无符号的对象进行运算和处理的实例。和之前一样，每行语句左边的数字是行标，仅作参考。

```
//测试有符号和无符号运算程序
1 module test_ v2001_ ops ()
2 reg [7: 0] a = 8'b01101111; //无符号数（十进制 111）
3 reg signed [3: 0] d = 4'b0011; //有符号数（十进制 +3）
4 reg signed [7:0] b = 8'10010110; //有符号数(十进制 -106)
5 reg signed [15: 0] c; //有符号数
6 initial
7 begin
```

```
8  c = a * b; //无符号数和有符号数相乘
9  #100;
10 c = $signed (a) * b; //2 个有符号数相乘
11 #100;
12 c = b / d; //有符号数相除
13 #100;
14 b = b >>> 4; //右移
15 #100;
16 d = d << 2; //逻辑左移
17 #100;
18 c = b * d; //有符号数相乘
19 #100;
20  $stop;
21 end
22 endmodule
```

时间	a	d	b	c	行数
0	111	3	-106	16650	8
100	111	3	-106	-11766	10
200	111	3	-106	-35	12
300	111	3	-7	-35	14
400	111	-4	-7	-35	16
500	111	-4	-7	28	18

代码 7.2　有符号和无符号数的运算

代码下方的表格是仿真运算模块 test_ v2001_ ops（）各个时段的结果，变量 a、b、c 和 d 都是以十进制来表示。

从第 6 行开始的时序语句模块以 initial 打头，且执行顺序和代码排列顺序一致，$stop 命令停止仿真。每一条语句的执行结果在表格里都有体现，并对应各自的行数。

代码第 8 行将一个有符号的数和一个无符号的数的乘积赋值给一个有符号的数。得出的结果是十进制 16650，原因在于其中一个乘积项是无符号的，系统将另一个有符号的变量转换为无符号数参与计算，即十进制 150，而不是十进制 -106。

第 10 行的语句在进行运算之前先将无符号的操作数 a 转换成有符号的数，因此所有的操作数均为有符号操作数，因而结果也就是有符号的（十进制 -11766）。

第 12 行的语句是有符号数之间的除法，十进制 -106 除以十进制 3，结果是十进制 -35。计算结果进行了四舍五入的取整处理。

第 14 行是有符号数的右移，使用的是带符号的操作符（>>>）。这种情况下，符号位（也就是最高位）被复制了 4 次，并且占据高四位，相当于将原来的操作数除以十进制 16 且保持原有的符号。用二进制来表示结果就是‘11111001’，也就是十进制的 -7。

逻辑左移的操作出现在代码的第 16 行。逻辑位移总是往空缺的位置添加逻辑 0，因此结果最终为‘1100’，即十进制 -4。

最后在第 18 行，两个负数相乘得到的结果为一个正数。

声明里用到的 signed，以及代码里出现的 $signed 和 $unsigned 的适用场合，是参与运算和处理的变量必须为 2 的补码，且数值必须有正负极性。本书的主要研究对象是有限状态机，其中的信号通常不是单比特就是多位总线，主要用来表示系统的状态。因此，带符号的各种运算后续将不再深入介绍。

对于寄存器型和线网型变量里出现的 z 或者 x，如果参与计算，结果的状态是未知态 x，下面给个例子来说明：

```
//给两个 4 位变量赋值
in1  = 4'b101x;
in2  = 4'b0110;
sum  = in1  + in2; //sum 的结果应该是 4'bxxxx，因为 in1 里有未知状态 x 的存在
```

图 7.11 是另一个用 Verilog 来描述按位逻辑操作的例子。代码第 4 行到第 7 行的连续赋值语句运用了逻辑与（&）、逻辑非（~）和逻辑或（|）等操作符。注意其中没有必要在输入取反的环节加入括号，因为逻辑非（~）本身的优先级就高于逻辑与（&）。但同样是逻辑与操作，它们之间的优先级就要用括号来区分了，因为逻辑与和逻辑或具有相同的优先级。

```
1  //4选1多路选择器，使用按位操作符
2  module mux4_to_1(output out,
3          input i0, i1, i2, i3, s1, s0);

4  assign out = (~s1 & ~s0 & i0)|
5          (~s1 & s0 & i1)|
6          (s1 & ~s0 & i2)|
7          (s1 & s0 & i3);

8  endmodule
```

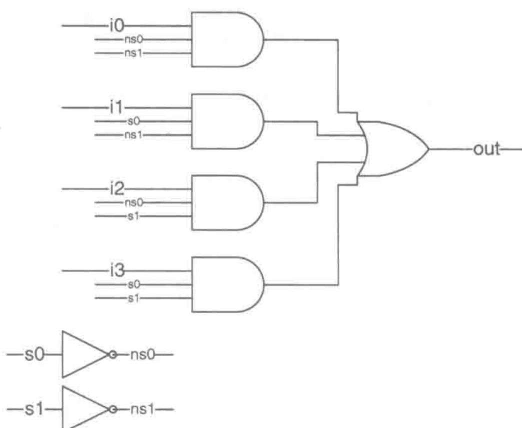


图 7.11 用按位操作符描述 4 选 1 多路选择器

图 7.12 是用另一种方法来诠释图 7.11。图中，嵌套条件操作符用来从 4 个输入 i0 到 i3 中选择一个输出，两个控制信号 s0、s1 的组合用来选择输入信号是否送到输出端 out。

对于条件操作符在用来进行嵌

套时是没有特别限制的，除非代码需要具备一定的可读性。

图 7.13 里的代码是另一个运用条件操作符的例子。代码第 3 行被用来描述‘三态缓冲器’，替代 8 个内置的名为 bufif1 的内置模块（见图 7.5）。

```
1  //4选1多路选择器，使用条件操作符
2  module mux4_to_1(output out,
3         input i0, i1, i2, i3, s1, s0);

4  assign out = s1 ? ( s0 ? i3 : i2)
5         : (s0 ? i1 : i0);

6  endmodule
```

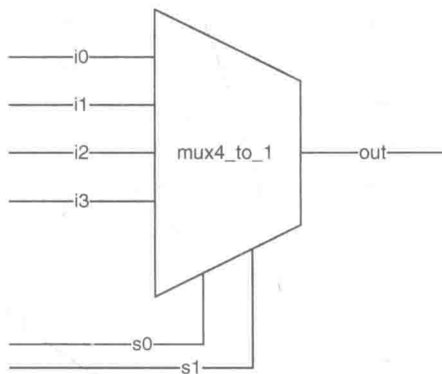


图 7.12 用嵌套条件操作符描述 4 选 1 多路选择器

当输入信号 enable 值为逻辑 1，Dataout 的输出结果和输入 Datain 的值一致；相反，当 enable 为逻辑 0，端口 Dataout 的值变为‘zzzzzzzz’。

端口方向除了输入（input）和输出（output），Verilog 还提供可以双向通信的端口，名为 inout。如图 7.14 所示，当双向端口作为输入时，如果没有数据进来，必须让其保持高阻状态 z，如代码第 3 行所示。Verilog 仿真器利用内置采样精度系统，来确定线网型变量被总线驱动时的值。这里，双向端口 Databi 通过外部信号驱动可以是逻辑 0，也可以是逻辑 1，因此它有两个驱动源。代码第 3 行表示 Databi 如果有输入值，则原来的高阻会被新的输入覆盖；因此在代码第 4 行，Datain 的赋值和加载到 Databi 上的一样。

```
1  //8位宽的三态缓冲器
2  module Tribuff8(input [7:0] Datain, input enable,
          output [7:0] Dataout);

3  assign Dataout = (enable == 1'b1)? Datain : 8'bz;

4  endmodule
```

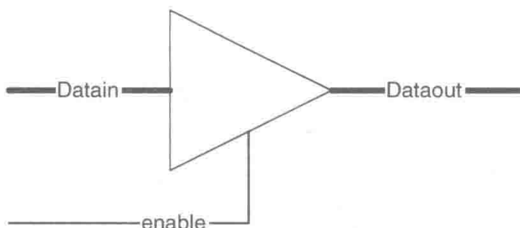


图 7.13 8 位三态缓冲器

```
1 //使用三态缓冲器的8位双向端口
2 module Bidir(input [7:0] Databuff,
             output [7:0] Datain,
             input enable, inout [7:0] Databi);

3 assign Databi = (enable == 1'b1)?Databuff : 8'bz;
4 assign Datain = Databi;

5 endmodule
```

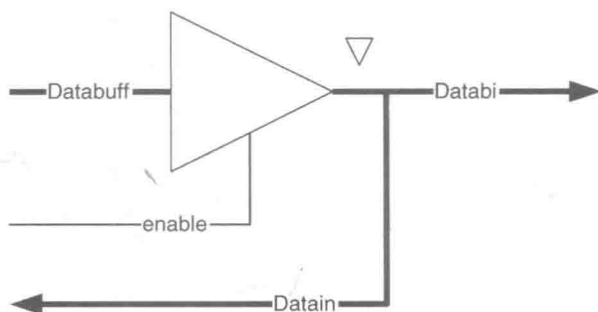


图7.14 双向总线构架

图7.15中描述的是运用逻辑或操作符(||)和等于操作符（==）来表达简易组合逻辑。模块的功能是一个三路输入的多数投票系统，当两个或两个以上的输入为逻辑1时，最终输出为逻辑1。

在代码第4行，输入信号被组合在一起，形成一个3位的线网型变量abc。组合操作符({})可以用来将任意一位或者多位的线网型变量或者寄存器型变量组成一个总线信号。本行代码把组合线网型变量以及连续赋值两个功能并成了一条语句。

第5行到第8行的连续赋值语句含有延迟信息，表示任何输入端的变化在输出端的体现必须经过10ns的延迟。这意味着不需要在模块内部增加额外的逻辑门来建立传输延迟。

从第5行开始，赋值语句右边使用了逻辑或（||）而不是按位或（|）。在这个模块里，使用哪一种逻辑操作符其实无所谓，但是某些情况下操作符选择按位还是按整体是很重要的，因为后者是基于布尔代数的理念。在Verilog里，多位数据任意组合除了全0，都被认为是有效的；下面举例说明：

```
wire [3: 0] a = 4'b1010; //有效
wire [3: 0] b = 4'b0101; //有效
wire [3: 0] c = a & b; //按位操作的结果是4'b0000，因此为低（无效）
wire d = a && b; //逻辑操作的结果是1'b1，即有效
```

Verilog HDL也支持一些比较操作符，例如大于、小于、大于等于等。图7.16

```
1 //带有10ns延迟的三输入多数投票器
2 `timescale 1 ns/ 1ns
3 module maj3(input a, b, c, output m);

4 wire [2:0] abc = {a, b, c}; //将输入合并

5 assign  #10 m = (abc == 3'b110)||
6                 (abc == 3'b101)||
7                 (abc == 3'b011)||
8                 (abc == 3'b111);  //输出信号m为逻辑1的所有输入组合

9 endmodule
```

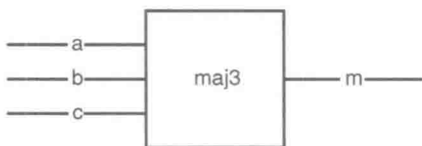


图 7.15 三路输入投票模块

是用上述比较操作符来描述一个存储芯片的地址解码单元。

图中模块的功能是激活一个 4 位低有效片选输出信号 Csbar [3: 0] 的其中一位，取决于输入端地址 16 位码字对应的十六进制的值。这种解码模块通常用来执行微处理器系统中的存储单元映射的功能。

第 4、6、8 和 10 行的连续赋值语句对应了输入端 Address 的不同赋值。例如，当输入端 Address 的值落在十六进制的 1500 ~ 16FF 的范围内时，Csbar [2] 的值为逻辑 0。

关系操作符的另一个应用实例如图 7.17 所示。它是一个用 Verilog 语言来描述的 4 位幅度比较器。

模块运用了关系操作符和对等操作符，并引入了其他的一些逻辑操作符，用于形成连续赋值语句右边的逻辑表达式，具体在代码第 5 行和第 7 行体现。注意第 行条件表达式里括号的运用方式，例如：((a > b) || ((a == b) & (agtbin == 1'b1)))?。

括号使得逻辑与在表达式中优先于逻辑或进行判断，整个式子的结果是“有效”或者“无效”，即 1'b1 或者 1'b0。条件表达式最终将逻辑 1 或者逻辑 0 赋给输出信号 agtbout。这里条件操作符 (?) 所构成的判断语句可以独立成为一行语句加入条件操作符纯粹为了向读者展示不同种类的操作符糅合在同一行语句里表现形式。

第 9 行连续赋值语句右边使用了按位逻辑操作符，这种情况下，输出的格式和逻辑操作符产生的结果是一样的，语句如下：altbout = (a < b) | ((a == b) & altbin)。

```
1  //16位地址解码单元
2  module Addr_dec(input [15:0] Address,
3                  output [3:0]  Csbar);

4  assign Csbar[0] = ~((Address >= 0) &&
5                     (Address <= 16'h03FF));
6  assign Csbar[1] = ~((Address >= 16'h0800) &&
7                     (Address <= 16'h12FF));
8  assign Csbar[2] = ~((Address >= 16'h1500) &&
9                     (Address <= 16'h16FF));
10 assign Csbar[3] = ~((Address >= 16'h1700) &&
11                    (Address <= 16'h18FF));

12 endmodule
```

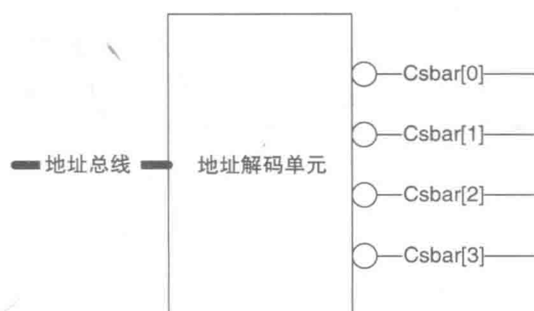


图 7.16 使用关系操作符进行地址解码

只要提供的变量不含有未知态（x）和高阻态（z），关系操作符以及逻辑操作符在上述实例中总是会得出逻辑 0 或者逻辑 1 的结果。如果操作数中含有上述不确定的情况，这些操作过程将产生未知态（x）的结果，具体例子如下：

```
reg [3: 0] A = 4'b1010;
reg [3: 0] B = 4'b1101;
reg [3: 0] C = 4'b1xxx;

A < = B //结果是逻辑 1
A > B //结果是逻辑 0
A && B //结果是逻辑 1
C = = B //结果是 x
A < C //结果是 x
C | | B //结果是 x
```

有时候，在测试或者仿真时，某个输出端的结果是未知或者高阻的情况是有意义的；因此，比较带有 x 或 z 的参数值具有一定的必要性。

Verilog HDL 提供了称之为全等于操作符（“= = =”和“! = =”）来达到这个目的。这种操作符可以比较未知态（x）和高阻态（z），也适用于普通的逻辑

```
1   //4位幅度比较器
2   module mag4comp(input [3:0] a, b,
3        input aeqbin, agtbin, altbin,
4        output aeqbout, agtbout, altbout);

5   assign aeqbout = ((a == b) && (aeqbin == 1'b1))?
6                   1'b1 : 1'b0;

7   assign agtbout = ((a > b) || ((a == b) &&
8            (agtbin == 1'b1))) ? 1'b1 : 1'b0;

9   assign altbout = (a < b) | ((a == b) & altbin);

10  endmodule
```

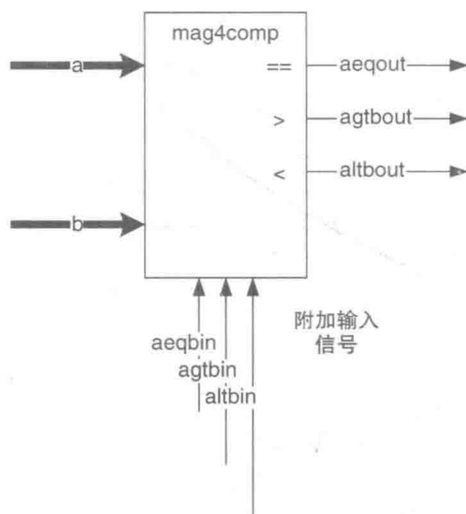


图 7.17　4 位幅度比较器

0 和逻辑 1 的比较，最终的比较结果总是逻辑 0 或者逻辑 1。下面举例说明：

```
reg [3: 0] K  =  4'b1xxz;
reg [3: 0] M  =  4'b1xxz;
reg [3: 0] N  =  4'b1xxx;

K  = = =  M //完全匹配，结果是逻辑 1
K  = = =  N //只有一位不同，结果是逻辑 0
M  !  = =  N //结果是逻辑 1
```

上面 3 个寄存器型（reg）变量的初始态都是未知的值；随后 3 组比较产生的结果是逻辑 0 或逻辑 1，按位比较过程中，每一位未知态将 4 种可能的情况（0，1，x，z）都纳入了考虑。

表7.3里最后一组操作符是复制和组合操作符，两种操作符都利用了大括号({})，这和C语言里用begin…end表示一组语句的格式类似。

组合({})操作符的作用是将新的变量附加在已有的后面，或者将多个变量组合起来，形成位数更多的变量。所有参与组合的操作数必须有固定的大小，即位数。组合方式可以是整个变量参与、部分参与或者其中的某一位参与，且这样的组合操作可以出现在赋值语句的左右任意一边或者两边同时出现。下面举例说明：

```
//A = 1'b1, B = 2'b00, C = 3'b110
Y = {A, B}; //Y 的值为 3'b100
Z = {C, B, 4'b0000}; //Z 的值为 9'b110000000
W = {A, B [0], C [1]}; //W 的值为 3'b101
```

下述Verilog语句描述在赋值操作符左边使用组合操作符。此时，代表“进位溢出”的线网型变量c_ out占据了5位结果的最高位，而操作符右边是两个4位变量和一个“进位位”三者的相加。

```
wire [3: 0] a, b, sum;
wire c_ in, c_ out;

//操作对象的长度是 5 位，即 [4: 0]
{c_ out, sum} = a + b + c_ in;
```

复制操作符可以用于本身，或者和组合操作符结合起来使用。复制操作符带有一个复制常数，用来标明被复制对象的操作次数。如果用j来表示被复制对象，k用来表示复制常数，那么操作语句的写法如下：

```
{k {j}}
```

下面给出一些涵盖了复制操作的语句：

```
//a = 1'b1, b = 2'b00, c = 2'b10
//单纯的复制操作
Y = {4 {a}} //Y 的值为 4'b1111
//复制和连接的操作
Y = {4 {a}, 2 {b}} //Y 的值为 8'b11110000
Y = {3 {c}, 2 {1'b1}} //Y 的值为 8'b10101011
```

一种可能的运用场合是复制带符号2的补码，例如将符号位进行扩展：

```
//2 的补码数值（十进制 -54）
wire [7: 0] data = 8'b11001010;

//向右移动 2 位
//移动后数值为 8'b11110010（十进制 -14）
assign data = {3 {data [7]}, data [6: 2]};
```

上述操作也可以用右移操作符‘>>>’来完成，然而如果用右移操作符的话，在线网型变量 data 的声明语句里就必须注明变量是带符号的。

7.3 Verilog HDL 操作符运用案例：汉明码编码器

本节将向大家介绍一个完整的工程案例，其中会用到之前介绍的部分 Verilog HDL 的操作符。图 7.18 是一个 8 位汉明码（Hamming code）[2] 编码器的结构框图和 Verilog HDL 代码。模块 Hamenc8 的功能是根据输入的 8 位字节数据产生一组奇偶校验数据；生成的校验数据随后和原始数据组成一个 13 位的汉明码字（Hamming codeword）[2]。产生的码字具有识别和纠正误码的功能，具体表现为系统可以纠正任何位置的某一位错误数据（包括校验位），并能够检测到任意两位的错误数据。

具体汉明码是如何检测和纠正误码的原理部分，留给有兴趣的读者自己去发掘，大家可以参考章节末尾给出的参考文献［2］。

```
1  //8位汉明编码器
2  module Hamenc8(input [7:0] Data,
3                 output [4:0] Parout);

4  //为每一位奇偶校验位定义用于异或计算的映射
5  localparam MaskP1 = 8'b01011011;
6  localparam MaskP2 = 8'b01101101;
7  localparam MaskP3 = 8'b10001110;
8  localparam MaskP4 = 8'b11110000;

9  assign Parout[4:1] = {^(Data & MaskP4),
                         ^(Data & MaskP3),
                         ^(Data & MaskP2),
                         ^(Data & MaskP1)};

10 assign Parout[0] = ^{Parout[4:1], Data};

11 endmodule
```



图 7.18　8 位汉明码编码器

代码第 2、3 行是定义模块端口的部分，输出信号 Parout 是一组 5 位奇偶校数据，通过将输入的 8 位数据 Data 的一部分进行异或操作获得。

代码第 5 行到第 8 行定义了一些局部参数，用于分别和输入数据进行异或操作。关键词 localparam 是用来定义专门用于某个模块内部使用的局部参数或者常量，即它们不受外界影响。例如，数据 MaskP1 定义了输入数据的子集，必须经过逻辑运算后产生 Parout [1]，过程如下：

```
Bit position - >76543210
MaskP1 = 8'b01011011;
Parout [1] = Data [6] ^Data [4] ^Data [3] ^Data [1] ^Da-
ta [0];
```

上述模 2 加法在模块里的实现的过程如下：首先将待处理的数据和 MaskP1 按位进行逻辑与（&）运算，然后将上一步逻辑与操作得出的所有逻辑 1 对应的数据位进行按位异或（^）操作。每一位奇偶校验数据的生成，都要经过这 2 步，具体代码见图 7.18 中代码第 9 行。

```
^ (Data & Mask P1)
```

组合操作符（{}）出现在连续赋值语句的右边，目的是通过重复上述步骤，并给 Parout 的高 4 位 Parout [4：1] 赋值。

检测任意两位误码的功能由 Parout 的最低位 Parout [0]（整体校验位）来完成。这一位生成方式是将输入数据（8 位）和之前所有的校验位（Parout [4：1]）进行模 2 加（异或）运算，在操作符运用方面包括组合操作符和按位异或操作符，具体代码参考第 10 行：

```
assign Parout [0] = ^ {Parout [4: 1], Data}。
```

上述语句经过按位操作并抵消将某些数据位从结果中剔除，原因在于如果同一位数据进行偶数次异或操作，其结果最终会是逻辑 0。因此数据的整体校验位赋值语句如下：

```
Parout [0] =Data [7] ^ Data [5] ^ Data [4] ^ Data [2] ^
Data [1] ^ Data [0]。
```

数据中的 Data [6] 和 Data [3] 没有参与运算。图 7.19 显示了完整的汉明码编码器奇偶校验输出从高到低每一位的逻辑运算原理图。这张图可以看成是逻辑综合工具通过图 7.18 的代码所进行的后期处理。

7.3.1　汉明码编码器的仿真

汉明码编码器的功能可以通过一系列实际操作来进行验证。比如将一组任意的数据作为输入，将系统输出的结果和事先根据编码原理计算的结果进行对比。

另一种比较系统化的方法，就是利用一个汉明码解码器来自动解码上述编码器的输出，这属于一种比较稳健的测试方法（当然这里前提是汉明码解码器工作正常）。

使用汉明码解码器，通过人为地导入一位或者两位误码，可以验证编码器的误码检测和纠正功能是否正常。

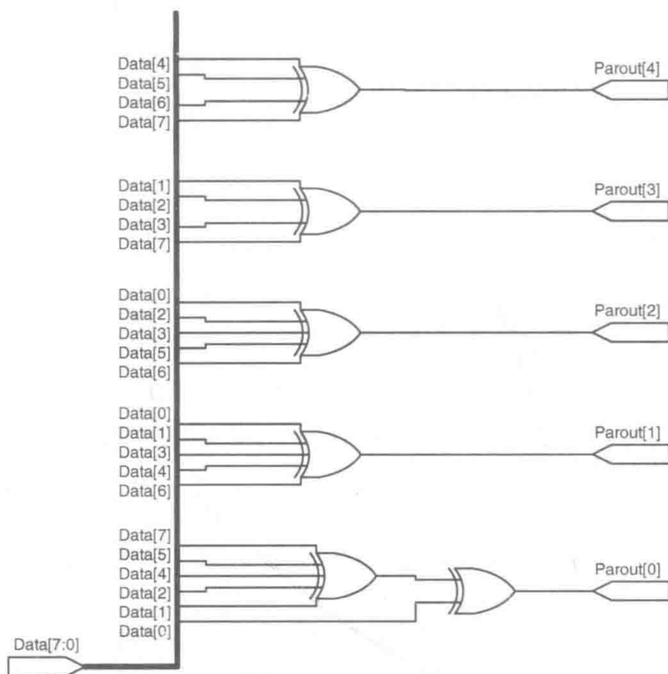


图 7.19 汉明码编码器逻辑框图

代码 7.3 是 8 位汉明码解码器源代码

```
//用 Verilog 描述汉明码解码器，13 位输入
1 module Hamdec8 (input [7: 0] Datain,
        input [4 : 0] Parin,
        output reg [7: 0] Dataout,
        output reg [4: 0] Parout,
        output reg NE, DED, SEC);

//定义产生每一位奇偶校验位的局部变量
2 localparam MaskP1  = 8'b01011011;
3 localparam MaskP2  = 8'b01101101;
4 localparam MaskP4  = 8'b10001110;
5 localparam MaskP8  = 8'b11110000;

6 reg [4:1] synd; //误码测试寄存器
7 reg P0; //重新产生的奇偶校验位
```

```
8 always @  (Datain or Parin)
9 begin

//导入默认的输出结果（假设数据没有误码）
10   NE  = 1'b1;
11   DED  = 1'b0;
12   SEC  = 1'b0;
13   Dataout  = Datain;
14   Parout  = Parin;
15   P0  = ^ {Parin, Datain};   //奇偶校验位
16
//产生特征误码
17   synd [4]  =  (^ (Datain& MaskP8)) ^ Parin [4];
18   synd [3]  =  (^ (Datain& MaskP4)) ^ Parin [3];
19   synd [2]  =  (^ (Datain& MaskP2)) ^ Parin [2];
20   synd [1]  =  (^ (Datain& MaskP1)) ^ Parin [1];
21   if ( (synd  = = 0) && (P0  = = 1'b0)) //没有误码
22    ; //接受默认输出
23   else if (P0  = = 1'b1) //单个误码（或者可能是奇数个误码）
24    begin
25       NE  = 1'b0;
26       SEC  = 1'b1;
//纠正单个误码
27        case (synd)
28         0:Parout[0] = ~Parin[0];
29         1:Parout[1] = ~Parin[1];
30         2:Parout[2] = ~Parin[2];
31         3:Dataout[0] = ~Datain[0];
32         4:Parout[3] = ~Parin[3];
33         5:Dataout[1] = ~Datain[1];
34         6:Dataout[2] = ~Datain[2];
35         7:Dataout[3] = ~Datain[3];
36         8:Parout[4] = ~Parin[4];
37         9:Dataout[4] = ~Datain[4];
38         10:Dataout[5] = ~Datain[5];
39         11:Dataout[6] = ~Datain[6];
```

```
40          12:Dataout[7] = ~Datain[7];
41       default:
42         begin
43           Dataout  = 8'b00000000;
44           Parout  = 5'b00000;
45         end
46       endcase
47     end
48   else if ((P0  = = 0) && (synd !  = 4'b0000))
49     begin //两位误码
50        NE  = 1'b0;
51        DED  = 1'b1;
52        Dataout  = 8'b00000000;
53        Parout  = 5'b00000;
54     end
55 end
56
57 endmodule
```

代码 7.3　8 位汉明码解码器代码示例

代码第一行定义了解码器的端口，13 位汉明码输入由 Datain 和 Parin 两个信号组成，处理后的码字输出端口为 Dataout 和 Parout。3 个诊断信号用来指示输入数据的处理结果：

NE：表示没有误码（Datain 和 Parin 经过系统的处理没有变化）；

DED：表示检测到两位误码（Dataout 和 Parout 都被设为全 0）；

SEC：表示有单个误码被纠正（如果出现多于两位的奇数个误码，系统将把这样的情况处理成只有一位误码，这时候最终的输出会出错）。

这里强调一下汉明码解码器的输出是寄存器型（reg），这是 Verilog 行为描述的特性决定的，即输出是 always 模块内部（从代码第 8 行开始）时序语句赋值的结果。

代码第 2 行到第 5 行所定义的局部变量和编码器模块里的局部变量声明方式相同，使用方法也类似，主要用来产生一个 4 位位宽、名为 synd 的信号（代码第 6 行）。这被称为特征误码。它和重新生成的整体校验位 P0（代码第 7 行）一起，成为确定汉明码内部误码的数量和位置的机制。

汉明码解码器的主体是一个 always 时序模块（从代码第 8 行开始）。语句由第 9 行的 begin 和第 55 行的 end 来整体引导，并根据输入端口 Datain 和 Parin 上面值的变化，按顺序依次执行。always 模块代表汉明码字解码的组合逻辑所对应的行为

描述。

在时序模块的开始，输出信号的赋值都设定为默认值，即代表“没有误码”的状态（代码第10行到第14行）。这表明时序语句所描述的是组合逻辑。随后从代码第15行到第20行，整体校验位和特征误码的产生机制、代码风格和编码器里的描述方式非常相似。

从第21行开始，条件语句对整体校验位和特征误码进行了测试，目的是当输入码字出现误码时系统能够及时上报。

如果整体校验位的值为0，特征误码的值也为全0，那么输入端进来的码字就是无误的，第22行的空命令（只有一个;）会让输出保持其默认值。

如果第一项 if... else 条件未能通过，那么下一个条件（第23行）将会测试是否存在单个误码。如果整体校验位的值为逻辑1，那么解码器会认为存在单个误码；第24行到第47行的代码针对上述判断会首先将“单个误码”的输出标识 SEC 信号拉高，然后再去纠正误码。纠正误码使用的是 case 语句（第27行到第46行），4位特征误码信号被用于定位误码的位置并将误码取反纠正。

最后一组 if... else 条件语句面对整体校验位是逻辑0和特征误码不全为0的情况，表明此刻输入数据里有两位误码。这时，汉明码解码器不能纠错，因此它只能报告“检测到两位误码”到输出端，并将 Dataout 和 Parout 端口全部清零。

汉明码编码解码系统的测试模块我们将其命名为 TestHammingCcts，如代码7.5所示，系统框图为图7.20。这里还需要一个模块，名为 InjectError，作用是将误码植入到汉明码编码器输出的数据里，然后再导入汉明码解码器。

InjectError 模块的代码如下表所示：

```
//向汉明码字中植入误码
1 module InjectError (input [7: 0] Din,
            input [4: 0] Pin,
            output [7: 0] Dout,
            output [4: 0] Pout,
            input [12: 0] Ein);
2 assign { Dout, Pout} = { Din, Pin} ^ Ein;
3 endmodule
```

代码7.4　13位误码植入模块

这段代码使用连续赋值语句将输入的13位汉明码其中的一位或者多位进行取反，具体方法是将输入数据和一个13位的误码数据 Ein 进行异或操作（代码第2行）。

对照图7.20和代码7.5，整个测试模块包含了编码器、解码器和误码植入单元等3个部分（第33行到第46行），还有两个 initial 时序模块 gen_ data 和 gen_ error，分别从代码的第13行和第20行开始。

```
//汉明码编码器和解码器测试固件
1 'timescale 1ns/1ns
2 module TestHammingCcts ();

//汉明码编码器数据输入
3 reg [7: 0] Data;
//误码模板
4 reg [12: 0] Error;
//汉明码编码器输出
5 wire [4: 0] Par;
//带误码的汉明码
6 wire [7: 0] EData;
7 wire [4: 0] EPar;
//汉明码解码器输出
8 wire DED;
9 wire NE;
10 wire SEC;
11 wire [7: 0] Dataout;
12 wire [4: 0] Parout;

13 initial //产生完整的测试数据
14 begin : gen_ data
15   Data = 0;
16  repeat (256)
17     #100 Data = Data + 1;
18    $stop;
19 end

20 initial //产生多个误码格式
21 begin : gen_ error
22   Error = 13'b0000000000000;
23   #1600;
24   Error = 13'b0000000000001;
25     #100;
26  repeat (100) //单误码反转
27     #100 Error = { Error [11: 0], Error [12] };
```

```
28  Error = 13'b0000000000011;
29    #100;
30  repeat (100) //双误码反转
31    #100 Error = { Error [11: 0], Error [12] };
32 end
//建立模块
33 Hamenc8 U1 (.Data (Data),
34            .Parout (Par) );

35 Hamdec8 U2 (.Datain (EData),
36            .Parin (EPar),
37            .Dataout (Dataout),
38            .DED (DED),
39            .NE (NE),
40            .Parout (Parout),
41            .SEC (SEC) );
42 InjectError U3 (.Din (Data),
43             .Ein (Error),
44             .Pin (Par),
45             .Dout (EData),
46             .Pout (EPar) );
47 endmodule
```

代码 7.5　汉明码编码/解码测试固件模块

名为 gen_ data 的 initial 模块使用 repeat 循环产生完整的 8 位输入数据，数值从十进制 0 到十进制 255，每 100ns 更新一个数值，代码运行到第 18 行时仿真通过命令 $stop 停止。代码第 20 行到第 32 行对应的 gen_ error 模块将信号 Error 初始化为全 0，并维持 1600ns，以便验证“无误码”的状态。

在第 24 行，Error 变为 13'b0000000000001，这样将汉明码编码的最低位设为误码所在的位置。等待 100ns 之后，用 repeat 循环（代码第 26 行和第 27 行）通过每隔 100ns 反转最高位和剩余的 12 位 100 次，这样每次在不同的位置产生单个误码，用来验证不管原始输入的数据值是什么，汉明码解码器是否可以纠正每一个位置上出现单个误码。

代码第 28 行到第 31 行将误码格式 Error 的值重新初始化为13'b0000000000011。表明这时候系统向汉明码中插入双误码。同样用反转的方法让双误码在不同的位置出现 100 次，每次间隔 100ns，用来验证解码器对于不同输入数据检测双误码的功能。仿真结果如图 7.21a ~ c 所示。

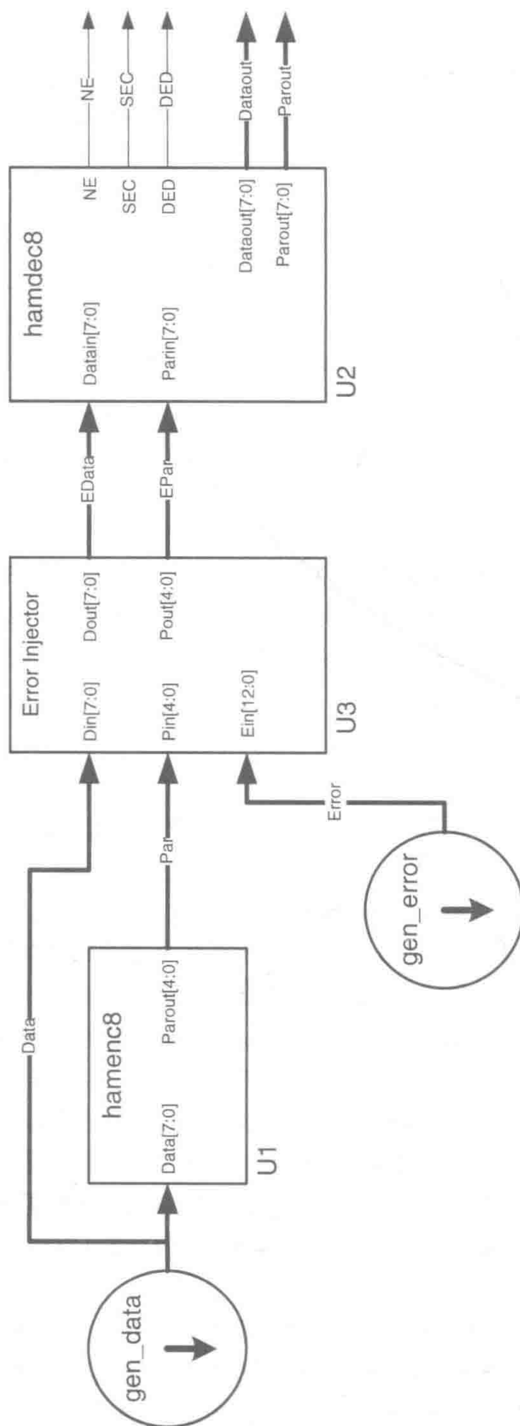


图 7.20　汉明码编码/解码测试固件框图

	0ns		200ns		400ns		600ns		800ns		1.0μs		1.2μs		1.4μs	
TestHammingCcts.Data[7:0]	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
TestHammingCcts.Par[4:0]	00	07	0B	0C	0D	0A	06	01	0E	09	05	02	03	04	08	0F
TestHammingCcts.Error[12:0]	0000															
TestHammingCcts.EData[7:0]	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
TestHammingCcts.EPar[4:0]	00	07	0B	0C	0D	0A	06	01	0E	09	05	02	03	04	08	0F
TestHammingCcts.SEC																
TestHammingCcts.NE																
TestHammingCcts.DED																
TestHammingCcts.Dataout[7:0]	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
TestHammingCcts.Parout[4:0]	00	07	0B	0C	0D	0A	06	01	0E	09	05	02	03	04	08	0F

1.6μs

a)

	4.0μs		4.2μs		4.4μs		4.6μs		4.8μs		5.0μs		5.2μs		5.4μs	
TestHammingCcts.Data[7:0]	28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34	35	36	37
TestHammingCcts.Par[4:0]	1B	1C	10	17	16	11	1D	1A	06	01	0D	0A	0B	0C	00	07
TestHammingCcts.Error[12:0]	0400	0800	1000	0001	0002	0004	0008	0010	0020	0040	0080	0100	0200	0400	0800	1000
TestHammingCcts.EData[7:0]	08	69	AA	2B	2C	2D	2E	2F	31	33	36	3B	24	15	76	B7
TestHammingCcts.EPar[4:0]	1B	1C	10	16	14	15	15	0A	06	01	0D	0A	0B	0C	00	07
TestHammingCcts.SEC																
TestHammingCcts.NE																
TestHammingCcts.DED																
TestHammingCcts.Dataout[7:0]	28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34	35	36	37
TestHammingCcts.Parout[4:0]	1B	1C	10	17	16	11	1D	1A	06	01	0D	0A	0B	0C	00	07

b)

		18.6μs		18.8μs		19.0μs		19.2μs		19.4μs		19.6μs		19.8μs		20.0μs
TestHammingCcts.Data[7:0]	B9	BA	BB	BC	BD	BE	BF	C0	C1	C2	C3	C4	C5	C6	C7	C8
TestHammingCcts.Par[4:0]	16	1A	1D	1C	1B	17	10	0F	08	04	03	02	05	09	0E	01
TestHammingCcts.Error[12:0]	0018	0030	0060	00C0	0180	0300	0600	0C00	1800	0009	0003	0006	000C	0018	0030	0060
TestHammingCcts.EData[7:0]	B9	BB	B8	BA	B1	A6	8F	A0	01	42	C3	C4	C5	C6		CB
TestHammingCcts.EPar[4:0]	0E	0A	1D	1C	1B	17	10	0F	08	05	00	04	09	11	1E	01
TestHammingCcts.SEC																
TestHammingCcts.NE																
TestHammingCcts.DED																
TestHammingCcts.Dataout[7:0]	00															
TestHammingCcts.Parout[4:0]	00															

c)

图 7.21　TestHammingCcts 仿真结果

a）无误码　b）单误码纠错　c）双误码检测

图 7.21a 为前 16 组测试数据，对应的误码格式是全 0（仿真结果第三行），即没有误码。图中前两行是 8 位数据（Data）和 5 位奇偶校验值（Par），组成了 13 位汉明码编码器的输出；所有的数据格式都是十六进制。

图中第 4 行和第 5 行是误码植入模块所对应的数据（EData 和 EPar），显示的结果和头两行是相同的，因为此时的情况是没有误码。诊断输出信号 SEC、NE 和 DED 相应的显示结果也证明了此刻汉明码中没有误码，最底下两行表示汉明码经过解码器之后的输出无变化。

图 7.21b 仿真了单误码的情况，同样图中第 3 行是误码格式对应的数值，相应

的位在出现误码时是由原先的逻辑 0 变为逻辑 1。

误码植入模块的输出（EData 和 EPar）和头两行的数据以及校验位总是相差一位（例如，在 4.2μs 时，十六进制 2A、10 变为十六进制 AA、10）。诊断输出信号 SEC、NE 和 DED 上面的值，也反映出输入的码字里有单个的误码，图中最下面两行显示了带有误码的数据，经过汉明码解码器被纠正后的结果（和头两行的原始数据相同）。

图 7.21c 对双误码情况进行了仿真，此时的误码格式里有两位数据是逻辑 1，如图第 3 行所示。

图中第 4、5 两行的误码植入模块的值（EData 和 EPar）和头两行的数据，以及校验位相差两位（例如，在 18.8μs 时，十六进制 BC、1C 变为 BA、1C）。诊断输出信号 SEC、NE 和 DED 反映了两位误码被检测到的情况。图 7.21c 最底部的数据输出对于两位误码的情况会产生全 0，因为一般情况下，含有两位误码的数据，其使用价值或者实际意义已经很小了。

本节用一个比较直观的实例——汉明码编码器/解码器及其测试模块来介绍 Verilog HDL 里各种常用操作符和不同的数据类型的使用方法。Verilog 语言里的行为描述，在设计相对复杂的组合逻辑系统中发挥了很大的作用。

第 8 章将介绍如何用 Verilog 语言设计时序逻辑系统，特别是 FSM。

参 考 文 献

1. Ciletti M.D. Modeling, Synthesis and Rapid Prototyping with the Verilog HDL. New Jersey: Prentice Hall, 1999.
2. Wakerly J.F. Digital Design: Principles and Practices, 4th Edition. New Jersey: Pearson Education, 2006 (Hamming codes: p. 61, section 2.15.3).

第 8 章　运用 Verilog HDL 描述组合逻辑和时序逻辑

8.1　描述数据流模式：回顾连续赋值语句

在上一章已经向大家介绍了各种用 Verilog 设计并描述的所谓数据流的例子。这种描述方式使用的是连续赋值语句的并行语句。连续赋值语句的执行流程是由语句左右两边的信号（通常是线网型 wire），通过触发事件来控制的，因此它们在描述组合逻辑时有独特的优势。这类语句一般可以通过关键词 assign 来辨识。在关键词后面会跟随一行或者多行赋值语句，用分号来结尾。

下面是几个描述组合逻辑的例子，它们都是运用的连续赋值语句：

```
//一些连续赋值语句示例
assign A = q [0], B = q [1], C = q [2];

assign out = (~s1 & ~s0 & i0) |
   (~s1 & s0 & i1) |
   (s1 &~s0 & i2) |
   (s1 & s0 & i3);

assign #15 { c_ out, sum} = a + b + c_ in;
```

操作符（=）的左边是线网型（wire）变量，右边是表达式。连续赋值语句的作用，就是在操作符左右两端建立一个静态连接。这意味着赋值语句是一直有效的，并且对语句中操作符右边任何信号（即输入信号）的变化做出反应。这些变化会导致表达式的结果发生变化，并更新操作符左边线网变量（即输出）的状态。通过这种方式，连续赋值语句几乎成了描述组合逻辑的专属语句。

之前也提到过，Verilog 模块里可以含有任意数量的连续赋值语句。它们可能出现在模块开头定义线网变量（wire）和寄存器变量（reg）部分，以及模块结尾 endmodule 关键词之间的任意地方。

在操作符右边的语句里，可以同时出现寄存器型（reg）和线网型（wire）变量，且操作符的使用也十分灵活。

而所谓的赋值目标（操作符左边）必须是线网型（wire）变量，因为它是被连续不断地驱动的。无论是单比特（1bit）还是多位（总线）变量，都可以成为连

续赋值语句的目标变量。

尽管不是很常见，但也是可以用连续赋值语句来描述时序逻辑的，以电平敏感锁存的形式出现。

图 8.1 中，条件操作符（?:）出现在赋值操作符的右边（代码第 2 行）。当信号 en 为高（逻辑 1）输出 q 被持续地赋予输入信号 data 上的值；当 en 变成逻辑 0，输出 q 被本身赋值，即反馈并维持 q 上面的值。从代码下方的逻辑图也可以看出这种关系。

需要注意的是，使用连续赋值语句来组成如图 8.1 所示的电平敏感锁存结构，相对来说并不多见。大部分逻辑综合软件在编译这一类设计构架时会提出警告。

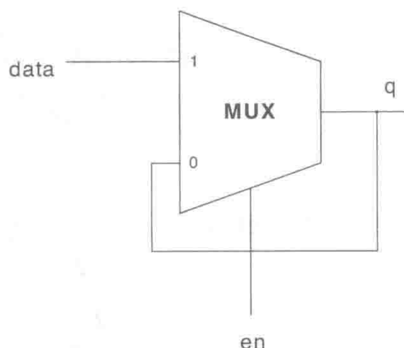


图 8.1　用连续赋值语句描述电平敏感锁存器

8.2　描述行为模式：时序模块

Verilog HDL 时序模块在硬件描述领域，定义了一种名为时序语句的代码格式。与传统的编程语言相似，这些语句的执行顺序和它们在书面上呈现的顺序是一样的。通过这种方式，时序模块可以提供一种构建具有行为模式或者算法的硬件描述方法。而且这种方式也让其成为比较理想的同步时序逻辑的描述语言，比较典型的同步时序电路是计数器和 FSM。然而，时序模块也可以用来描述组合逻辑。

读者会发现一些常用的 Verilog 时序语句和 C 语言里的一些语句有很多相似的地方。除了下面即将讨论的两种主要的时序语句模块，Verilog HDL 还拥有被称为 task 和 function 的两个专门用于执行时序功能的函数。但它们不在本书的讨论范围内，有兴趣的读者可以阅读本章节最后的参考文献［1］。

Verilog HDL 里的时序模块主要有以下两种：

一个是 always 模块。模块里的语句是不断重复地执行的，重复执行的条件一般和所谓的触发机制有关。一个 always 模块在运行时，看起来更像是一个永久循环。它可以用来描述任何类型的数字电路。

另一个是 initial 模块。这种时序模块从头到尾只执行一次，且在仿真的 0 时段开始执行。Verilog 里的 initial 模块几乎是专门为了仿真的测试固件设立的，通常用来定义输入端的触发信号，并控制整个仿真的执行时段。尽管仿真模板里很有可能出现 initial 模块，用来初始化各种变量或者加载数据，但它一般很少用来设计并描述硬件电路。

上述两种时序模块实际上都属于并行语句格式。因此，它们可以在硬件模块中出现的数量是不受限制的。always 和 initial 语句模块在整个代码中出现的位置和

序，不会影响到它们各自的执行。从这个角度看，时序模块和连续赋值语句有一定的相似性：后者是根据赋值操作符右边的输入信号不断变化，更新当前目标变量的值或者状态，而前者是根据某个触发条件来执行一系列的语句。

图 8.2 前半部分是 initial 时序模块的语法格式，随后通过一个例子向读者展示如何运用这样的模块来产生一个时钟信号。

代码第 3 行到第 8 行是一个完整的 initial 语句模块，其中包含的语句可以是一行，也可以是多行，用 begin…end 来引导。有时候，如果 initial 模块中的语句只有一行，这时候从语法角度来说是允许省略 begin…end 的引导格式的，正如代码第 12、13 行所示。当然，我们提倡不管是单行还是多行语句，大家都把引导格式加上，以免一时疏忽出现语法上的错误。

图 8.2 的后半段用 initial 模块引导一段代码（第 14 行到第 21 行），目的是产生一个持续不断的时钟信号。局部变量 PERIOD 的声明在第 14 行，且将时钟的周期设置为 100 个时间单位。initial 模块的执行从 0 时段开始（第 18 行），信号 CLK 被初始化为逻辑 0，这里 CLK 信号必须被定义为寄存器型变量（reg），因为它在被赋值的同时，还需要保持当前的赋值到下一个输入变化的到来。还需要说明的是信号 CLK 的初始化（代码第 15 行）可以和其定义的声明结合在一起，代码格式如下：`reg CLK = 1'b0`。

在定义了 CLK 信号之后，initial 模块将在第 19 和第 20 行继续执行产生时钟的功能。这里有一个 forever 循环，意思是永久循环，其语法格式如下：

```
forever
begin
  //时序语句 1
  //时序语句 2
  …
end
```

和 initial 模块一样，forever 循环里包含的语句数量也是不受限制的，并且能够永久重复执行。如果语句多于一行，也必须加入 begin…end 来引导。第 20 行的时序语句里还加入了系统延迟（语句中#后面表示一定时间的延迟）。而此条语句的作用是将 CLK 信号的状态，每隔 50 个单位时间取反，并重复这个作用。这样 CLK 信号的输出状态就如图 8.2 底部的效果一样了。

实际操作过程中，图 8.2 中的 Verilog 代码所描述的功能（第 14 行到第 21 行），在仿真时会有潜在的问题，因为目前大部分的仿真软件在内部会默认程序被设定为永久运行，或者带有类似的命令（例如 Modelsim® 里的 “run - all” 命令）。因此第 19 和第 20 行所出现的 forever 循环会让仿真器无法停下来，或者直到主机将所有存储空间填满了仿真数据后，自动内存溢出。

因此解决上述问题的办法有两种：

```
1  //initial时序模块的基本语法格式
2  //会有多条语句
3  initial
4  begin
5     // 时序语句1
6     // 时序语句2
7     ...
8  end
9
10 //initial时序模块的基本语法格式
11 //只有1条语句(无需begin...end)
12 initial
13    //时序语句


14 localparam PERIOD = 100;  //时钟周期

15 reg CLK;

16 initial
17 begin
18   CLK = 1'b0;
19   forever  //无限循环
20     #(PERIOD/2) CLK = ~CLK;
21 end
```

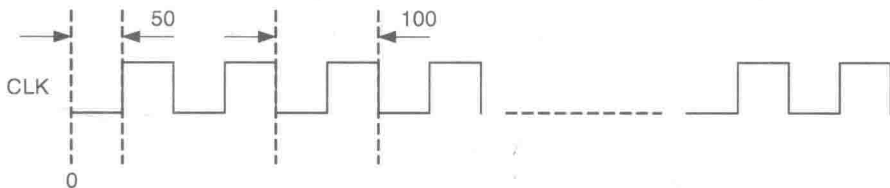


图 8.2　initial 模块的语法格式和示例

1）加一个新的 initial 模块，并用 $ stop 系统命令在适当时让仿真器停下来；

2）将 forever 循环替换成 repeat 循环。

第一种方法所需要添加的模块语句写法如下：

```
//n 代表了系统需要产生的时钟信号的周期数
initial # (PERIOD *  n)  $stop;
```

上述语句可以加在图 8.2 中第 14 行代码之后的任意地方，由于原有代码第 1 行开始的 initial 模块和新加入的 initial 模块，在执行时间上是同步的，也就是说两个模块都在 0 时段开始执行。因此，延迟后的 $ stop 命令所执行的时刻，正好和系统所输出的时钟周期数吻合，即 n * PERIOD 秒，也就是说在原来 initial 模块产生 n 个时钟周期后停止输出。反映到仿真器的界面上，看到的结果就是仿真一共持续 n 个时钟周期。要注意的是，为了编译通过，这里的变量 n，必须提前用一个

数替代或者定义成一个局部参数。

第二种方法涉及修改 initial 模块里的语句（代码第 16 行到第 21 行），新的代码如下：

```
1 initial
2 begin
3   CLK = 1'b0;
4  repeat (n) //重复执行固定的循环
5 begin
6   # (PERIOD / 2) CLK = 1'b1;
7   # (PERIOD / 2) CLK = 1'b0;
8 end
9  $stop;
10 end
```

其中的 repeat 循环是一个时序模块，作用是让一行或多行语句执行一定数量的循环。上述代码里，n 定义了仿真过程中所需要的时钟周期数。循环主体语句里，还带有 2 个延迟附加在寄存器变量 CLK 上，同时 begin... end 的框架也是需要的。

每一个 repeat 循环持续 100 个时间单位，即一个周期。一旦所需要的时钟周期数产生完毕，repeat 循环将自动终止，且仿真也到此结束，系统将执行第 9 行停止命令。

对于 repeat 和 forever 两种循环来说，需要在此指明的一点，就是两者都不能被综合成硬件电路。因此，这些语句只能用在 Verilog 的测试固件或者仿真模块里。

代码 8.1（a ~ e）给出 Verilog HDL 里 always 时序模块所对应的不同格式。最通俗的是代码 8.1 里的 a 格式：关键词 always 后面紧跟着的是条件表述；它决定了下面的时序语句在什么情况下开始执行。其中的@（event expression）对于组合逻辑和时序逻辑的描述都是必要的。

和 initial 模块一样，begin... end 的引导构架可以在 always 模块内只有一条时序语句时被省略。代码 8.1 里的 e 格式反映了这种情况。

```
a)
1 always @  (event_ expression)
2 begin
3   //时序语句 1
4   //时序语句 2
5   …
5 end

b)
```

```
1 always @  (input1 or input2 or input3...)
2 begin
3    //时序语句 1
4    //时序语句 2
5    …
6 end
```

```
c)
1 always @  (input1, input2, input3...)
2 begin
3    //时序语句 1
4    //时序语句 2
5    …
6 end
```

```
d)
1 always @  (* )
2 begin
3    //时序语句 1
4    //时序语句 2
5    …
6 end
```

```
e)
1 always @  (a)
2    y = a *  a;
```

代码 8.1 always 时序模块的不同格式

a）always 时序模块的通用模式；b）用或（or）操作符区分条件的 always 时序模块；c）用逗号区分条件的 always 时序模块；d）用通配符作为条件的 always 时序模块；e）只有一行时序语句的 always 时序模块。

和 initial 模块不一样的地方是，只要符合后面的条件表述，always 模块里的时序语句是重复执行的。当里面的每条语句执行结束后，程序执行进程会返回到模块里的第一行语句并在此处等待（挂起）。当条件表述再一次符合时，时序语句将又一次开始执行。条件表述的特性决定了被描述的逻辑电路的特性。作为常规的编程指南，代码 8.1 里的任何一种格式都可以用来描述组合逻辑。然而，代码 8.1b 格式大部分情况下用来描述时序逻辑，但是需要一些改进（后面会讲到）。

另一个和 initial 模块相同的特性是，always 模块的内部信号必须定义成寄存器型（reg）变量，因为所有的信号在下一次执行条件满足之前必须维持当前的状态。

有时候 always 模块可以用来替代使用 initial 模块时，需要依靠 forever 循环来实现的功能。例如，图 8.2 里产生时钟波形的测试程序，用 always 模块表述的方法如下：

```
1 localparam PERIOD = 100; //时钟周期数

2 reg CLK = 1'b0;

3 always
4 begin
5   #(PERIOD/2) CLK = 1'b1;
6   #(PERIOD/2) CLK = 1'b0;
7 end
```

上述语句的第 3 行到第 7 行用 always 模块来实现时，不需要条件表述来触发。因为其中每一条时序语句的执行时间点，都是通过延迟来控制的。

这个例子显现了 always 模块的一个重要特性：程序的运行条件是，模块内至少有一条带有延迟特性的语句来控制所有语句的执行时间点；或者当模块符合条件表述的要求时，所有语句按顺序全部执行。如果上述两种情况同时出现，会产生混淆，因此是不允许的。

always 模块里面如果没有出现任何控制程序执行时间点的机制，仿真工具会报错，并提示描述语句含有零延迟的无限循环，导致仿真器不能正常运行，“卡”在零时刻（仿真的起始点）。

总之，在测试模块里是不推荐使用 always 模块的，因为需要区分哪些模块是专门用来仿真的，哪些是专门用来设计并综合的。

8.3　时序语句模块：阻塞和非阻塞

对于 always 模块来说，其中的语句执行条件是信号的值发生变化，满足条件表述。到下一次满足条件并执行语句之前，模块处于一个静止状态；所以，模块内部的所有信号必须能够保持当前的值，或者说上次执行时被赋予的值。换一种说法就是模块内部的信号不是被永久驱动的。这和之前提到的所有变量必须是寄存器型（reg）的说法是一致的，并且它们只能出现在时序赋值语句的左边。

上述制约的对象，并不适用于那些出现条件表述里的变量。不过，触发时序模块的变量类型可以是寄存器型（reg），也可以是线网型（wire）。这意味着模块的输入信号、逻辑门的输出，以及连续赋值语句都能够成为触发时序语句模块的条件。因此，在硬件设计中可以随意地混合运用行为描述和数据流描述等不同类型的语句。

8.3.1 时序语句

表 8.1 列出了最常用的时序语句，它们经常出现在 initial 或者 always 模块里，有些和 C 语言里使用的格式很相像，而其他的是 Verilog HDL 专属格式。

本节将不会对时序语句的语义进行详细阐述，取而代之的是会用具体的代码实例来说明如何使用它们。不过表 8.1 并没有列出所有的时序语句，一些不太常用的结构，例如并行模块（fork…join）和进程连续赋值语句等，将留给有兴趣的读者自己去发觉[1]。

表 8.1 中方括号（[ ]）里的内容是备选的，大括号（{ }）里的内容可以重复出现，所有粗体的关键词必须以英文小写字母出现。

表 8.1　常用 Verilog HDL 时序语句

时序语句	语句说明
=	阻塞时序赋值
< =	非阻塞时序赋值
;	空语句。并且也出现在每一行时序语句后面
begin {时序语句} end	模块语句或者复合语句。多条时序语句必须使用 begin…end 引导
if（表述对象） 时序语句 [else 时序语句]	条件语句，其中的“表述对象”必须带有括号，else 部分不是必需的，并且条件语句可以内部相互套嵌。多条时序语句必须用 begin…end 来引导
case（表述对象） { {数值,}：时序语句} [default：时序语句] endcase	多路选择语句，其中的“表述对象”必须带有括号。“数值”可以有多个分支但不能有交集。如果所列出的“数值”没有覆盖所有可能性，必须运用“默认（default）”分支以增加代码完整性。多条时序语句需要用 begin…end 引导
forever 时序语句	无条件永久循环。多条时序语句必须用 begin…end 引导
repeat（表述对象） 时序语句	有限循环，其中的时序语句执行的次数和表述对象所对应的数值一样。多条时序语句必须用 begin…end 引导
while（表述对象） 时序语句	只要表述对象的值非零，测试循环（和 C 语言一样）就会一直运行。多条时序语句必须用 begin…end 引导。
for（表述对象 1；表述对象 2；表述对象 3）时序语句	通用循环结构（和 C 语言一样）。多条时序语句必须用 begin…end 引导
#（时间数值）时序语句	在执行时序语句之前模块会暂停（延迟）一段时间，时间数值表示暂停的时间单位数量
@（表述事件）时序语句	时序语句执行的条件是表述事件被触发

连续赋值语句在并行执行时，使用的操作符只有“ = ”一种。根据表 8.1，时序语句有两种赋值方式：

- 阻塞赋值——使用“ = ”操作符；
- 非阻塞赋值——使用“ < = ”操作符。

这两者的区别非常微妙，如果不能很好地理解而导致使用不当，会让仿真或者综合出现问题。

阻塞赋值语句在描述组合逻辑时，是比较常用的时序赋值语句形式。正如其名，它的特点是赋值对象的状态，在下一条语句被执行之前就已经更新了，和传统的编程语言比较类似。换句话说，一条阻塞语句在执行完毕之前是不允许（阻挡）下一条语句执行的。阻塞语句的另一个特性就是对于同一个信号，后一条语句都会有效地覆盖前面语句执行的结果。典型的例子是第 7 章介绍的汉明解码器（代码 7.3），解码器的所有输出信号在执行仿真程序前，都被初始化到默认状态。

对于非阻塞语句，仿真器在下一个仿真周期到来之前执行所有语句，这一系列动作通常发生在时序模块的结尾（或者在模块的执行条件下一次得到满足之前）。这时候，前后语句之间无法形成制约，而且所有的语句是在同一个时间点执行的。

在同一个时钟的驱动下，非阻塞语句可以用来给多个寄存器型（reg）变量同时赋值。图 8.3 是一个比较典型的例子。

代码第 17、18 和 19 行所对应的 3 条非阻塞语句，在名为“CLK”的信号的上升沿同时执行。代码第 15 行的条件表述里运用了 posedge（上升沿 positive - edge 的英文缩写）这个限定词，即 always 时序模块会在信号 CLK 的状态从逻辑 0 变为逻辑 1 时触发并执行。这种特定的触发方式在描述同步时序逻辑时被广泛采用，会在后续章节做进一步详细说明。

根据时序模块内部非阻塞语句的本质，例如在第一个时钟上升沿到来时，赋给 R2 的值是当前 R1 的值，即“未知态”（1'bx）。同样，在第二个时钟上升沿到来时，R3 上的值也是未知态 x，即 R2 的当前值 1'bx。所以，R1、R2 和 R3 初始的未知态，在 3 个时钟脉冲之后都变成逻辑 0。用这种方式，非阻塞语句在这里所描述的内容，事实上相当于一个 3 位的移位寄存器，如图 8.4 所示。

图 8.5 和图 8.3 几乎一样，除了代码的 17、18 和 19 三行，这里它们变成了阻塞语句。寄存器变量 R1、R2 和 R3 的初始值仍然是未知的，且寄存器变量 R0 在零时刻被初始化到逻辑 0。

图 8.5 的仿真结果充分说明了阻塞语句的作用：3 个信号都在时钟第一个上升沿到来时变为逻辑 0。这是因为阻塞语句在模块执行后面的语句之前，就将变量的值及时更新。结果就是 3 条语句的效果，相当于一条将 R0 的值直接赋给 R3 的语句。always 模块的等效电路如图 8.6 所示。

至于在时序模块里到底是使用阻塞语句还是非阻塞语句，取决于所描述的数字

```
1  `timescale 1 ns/ 1 ns
2  module non_blocking_assignmnts();

3  reg R1, R2, R3, R0, CLK;

4  initial
5  begin
6      R0 = 1'b0;
7      CLK = 1'b0;
8      repeat(3)
9      begin
10         #50 CLK = 1'b1;
11         #50 CLK = 1'b0;
12     end
13     $stop;
14 end

15 always @(posedge CLK)
16 begin //一组非阻塞语句
17     R1 <= R0;
18     R2 <= R1;
19     R3 <= R2;
20 end

21 endmodule
```

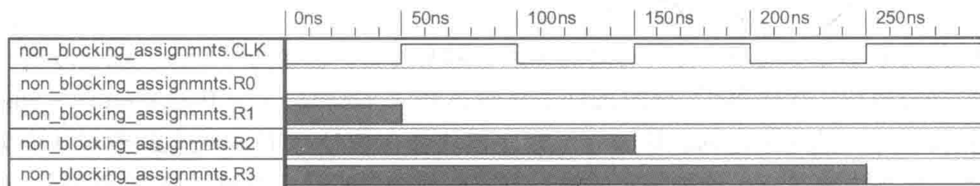


图 8.3　非阻塞语句的说明

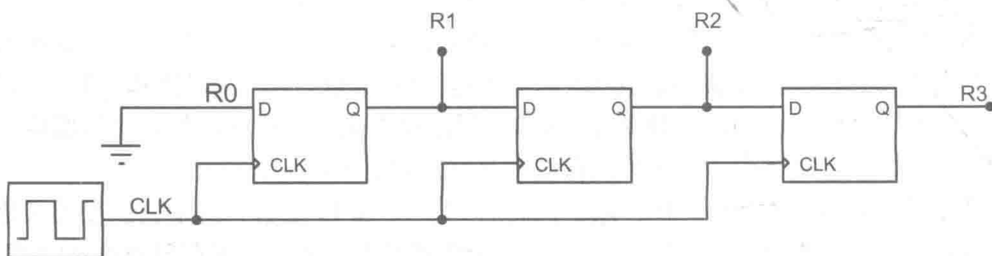


图 8.4　非阻塞语句等效电路

逻辑电路的特性。一般来说，在描述同步时序逻辑时，建议使用非阻塞语句；而对于组合逻辑，使用阻塞语句比较合适。

在测试平台上，时序模块一般用 initial 来引导，所以阻塞语句在这里比较常见。

```
1    `timescale 1 ns/ 1 ns
2    module  blocking_assignmnts();
3    reg  R1, R2, R3, R0, CLK;
4    initial
5    begin
6        R0 = 1'b0;
7        CLK = 1'b0;
8        repeat  (3)
9        begin
10           #50 CLK = 1'b1;
11           #50 CLK = 1'b0;
12       end
13       $stop;
14   end
15   always @( posedge CLK )
16   begin //一组阻塞语句
17       R1 = R0;
18       R2 = R1;
19       R3 = R2;
20   end
21   endmodule
```

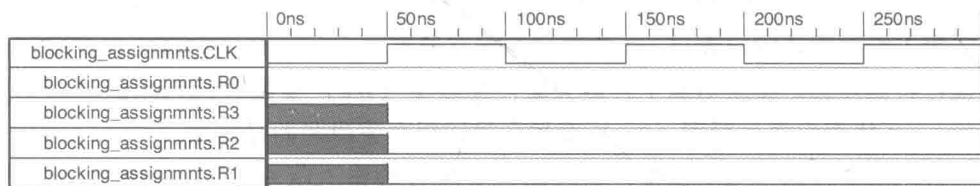


图 8.5　阻塞语句的说明

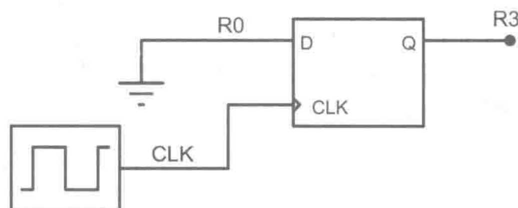


图 8.6　阻塞语句等效电路

对于上述内容还有一点要提醒的是，阻塞语句和非阻塞语句在同一个时序模块里不要同时出现。

8.4　用时序模块描述组合逻辑

各种不同类型的时序语句所组成的时序模块，可以用来描述几乎任意类型的数字逻辑电路。图 8.7 给出一个用 always 时序模块来描述的多路选择器。

```
1 module mux(output reg out, input a, b, sel);

2 always @(a or b or sel)
3 begin
4   if (sel)
5     out <= a;
6   else
7     out <= b;
8 end
9 endmodule
```

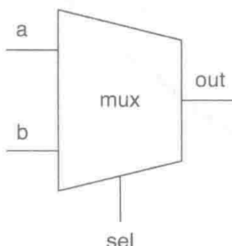


图 8.7　用 always 模块描述两路输入的多路选择器

第 1 行声明了输出端口 out 的类型为寄存器型（reg），而且它也出现在赋值语句操作符的左边。模块里除了告诉大家寄存器型（reg）变量在代码里的缩写形式是 reg，还体现出寄存器型（reg）变量在描述组合逻辑时的必要性。

代码第 2 行括号里的条件，表述涵盖了所有的输入信号，并用关键词 or 来隔开。这种格式和 1995 年最初的 Verilog 风格一样。而最新版本的语法，允许将任意出现在赋值语句操作符右边的寄存器型（reg）或者线网型（wire）的变量，用逗号或者通配符“＊”隔开。

不管条件表述的格式是什么，其意义都是一样的，即任何输入端的变化，都将触发模块内的语句按顺序执行。

从第 5 行到第 7 行的时序语句，使用了前面章节推荐的非阻塞格式。对输出信号 out 的赋值，要么是输入 a，要么是输入 b，取决于输入选择信号 sel 的状态。

使用 always 时序模块来描述组合逻辑的另一个特性，就是有可能产生不完整赋值的现象。这种情况出现在当使用 if…else 结构时，会省略最后一个 else 部分，导致被赋值的寄存器型（reg）变量，可以保持上一次被赋予的状态。

从硬件综合的角度看，这种不完整的赋值会形成一种类似锁存的状态。这种现象偶尔会成为设计者的本意，然而多数情况下是由于设计过程中，不小心忽略了最

后一个 else 的部分，或者忘记给输出端赋予一个默认的值。不管是什么原因，逻辑综合工具在遇到这种情况时，都会给出一个警告。

下面把使用 always 模块描述组合逻辑的要点总结如下：

使用代码 8.1b ~ d 里的其中一种格式，将所有组合逻辑对应的输入体现在条件表述里。

为了避免产生不必要的锁存状态，请使用下面两种方法中的一种：

——在 always 模块开端，使用例如 if、case 等时序结构之前，将所有的输出信号设定默认值；

——如果没有设定默认值，务必确保所有输入端的变化，在输出端都有对应的赋值。

图 8.8 给出了一个关于上述不完整赋值现象的示例。

```
1  module latch_implied(input a, b, c,
2                       input [1:0] sel,
3                       output reg y);
4  always @(*)//通配符触发
5  begin
6    if (sel == 2'b00)
7      y = a;
8    else if (sel == 2'b01)
9      y = b;
10   else if (sel == 2'b10)
11     y = c;
12 end
13 endmodule
```

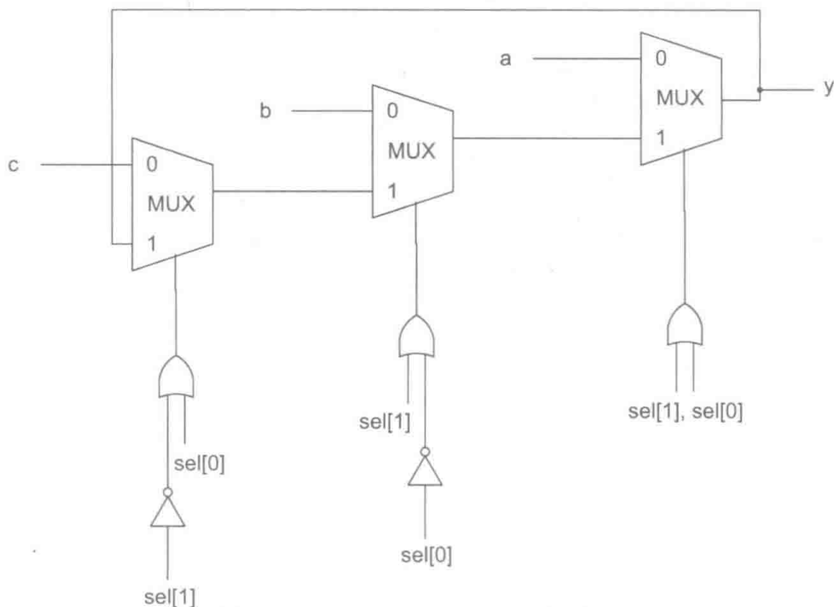


图 8.8　不完整赋值所衍生的锁存效应

latch_ implied 模块使用了 always 模块语句，来描述选择电路的行为模式。输入信号 sel［1：0］负责选择 3 个输入信号 a、b 和 c 的其中一个，作为输出端 y 的赋值。

这其中有一个假设，就是输出信号 y 的值，在 sel 为 2'b11 时是逻辑 0。根据源程序，这个显然是不正确的，因为缺少一个 else 的分句，y 将保持它的当前值（因为它是寄存器型）。软件在综合设计时，从 if…else 结构和寄存器型变量中，“推断”出这里需要一个锁存的功能。因此，在电路里出现了输出端 y，同时被反馈到第一个多路选择器的其中一个输入端的现象。

如果要去掉图 8.8 里电路所带的锁存功能，有两种不同的方法。图 8.9a 和 b 给出了答案，并在图 8.9c 部分给出最终正确的电路图。

```
1   module data_selector(input a, b, c,
2                  input   [1:0] sel,
3                  output reg    y);
4    always @(a, b, c, sel) // 同 '*'
5    begin
6      if (sel == 2'b00)
7        y = a;
8      else  if  (sel == 2'b01)
9        y = b;
10    else  if  (sel == 2'b10)
11       y = c;
12    else
13       y = 1'b0;
14   end
15   endmodule
```

a)

```
1    module data_selector(input a, b, c,
2                  input [1:0] sel,
3                  output reg y);

4    always @(a or b or c or sel)
5    begin
6      y = 1'b0; // 默认赋值
7      if (sel == 2'b00)
8        y = a;
9      else if (sel == 2'b01)
10       y = b;
11     else if (sel == 2'b10)
12       y = c;
13   end
14
15   endmodule
```

b)

图 8.9　取缔不必要的锁存反馈

a）增加一条 else 分句　b）将输出设定默认值

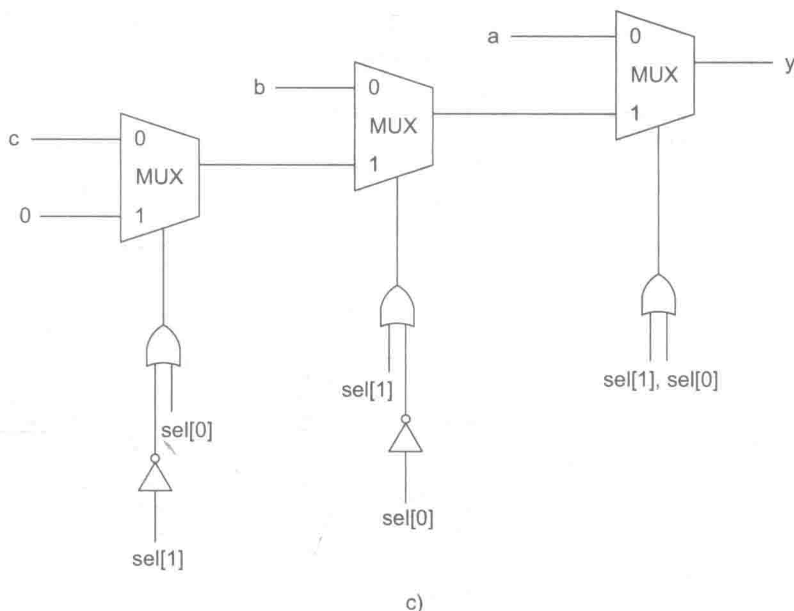


图 8.9　取缔不必要的锁存反馈（续）

c）a 和 b 对应的综合电路

图 8.9a 的代码在第 12 行和第 13 行加了一个 else 分句，这样就把输出信号 y 所对应的所有输入情况全部涵盖了。而图 8.9b 的代码在第 6 行给 y 设定了一个默认值逻辑 0，也达到了同样的效果。

对比两种解决锁存效应的方法，第二种设定默认值的方法似乎更加直观，因此，推荐使用第二种方法。

下面一个例子是关于如何运用时序模块来描述组合逻辑。第一个如图 8.10 所示，描述一个 3 路输入、8 路输出的解码器（类似像 74LS138 这样的 TTL 器件）。

ttl138 模块的功能是解码一个 3 位的输入信号（A，B，C），然后激活对应的 8 个低有效输出的其中一个。解码过程由 3 个输入信号进行控制（使能），它们分别是（G1，G2A，G2B），3 个信号的值必须依次设为 1、0、0。如果使能输入信号的值不是 1、0、0 的组合，输出端 Y 的所有位都被拉高。

描述上述行为的是一个 always 时序模块，它对所有输入端的变化都能做出响应，图 8.10 里代码的第 3 行是模块的开端。输出端 Y 在代码第 5 行被设定了默认值为全 1，随后用 if 语句来判断各种条件并把其中的一位拉低，具体判定是根据 A，B，C 的组合所对应的十进制数的值（代码第 6、7 两行）。

ttl138 的仿真如图 8.11 所示，测试固件运用了所谓的被命名的时序模块（代码第 6 行）。模块的名字是 gen_ tests，它属于一种标记，如果需要使用必须放在关键词 begin 的后面，用冒号引出。通过这样的方式给时序模块命名（always 和 initial 模块都适用）让其中的变量，例如寄存器型（reg）和整型（integer）的声明和运

```
1 module ttl138(input A, B, C, G1, G2A, G2B,
2              output reg [7:0] Y);

3 always @(A, B, C, G1, G2A, G2B)
4 begin
5    Y = 8'hFF; //设定默认输出
6    if (G1 & ~G2A & ~G2B)
7          Y[{A, B, C}] = 1'b0;
8 end

9 endmodule
```

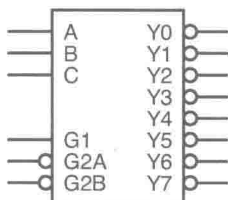


图 8.10　3－8 译码器 Verilog 描述代码和框图

用都限定在模块范围内。这些局部定义的对象如果要在外部调用，必须将模块的名称写在前面。例如，整数 t 在图 8.11 里的代码中，如果要在 initial 模块外部被调用，必须写成 gen_ tests. t

运用局部声明的对象，可以让描述更加结构化。然而，需要注意的是，不是所有的综合软件工具都能识别这种格式。

initial 模块里的整数 t 用来控制 for 循环的周期（代码第 9 行到第 12 行）。循环的作用是向解码器提供一组完整的 A、B、C 三个输入信号的值（十进制 0～7）。Verilog 的 for 循环的语法和格式与 C 语言里非常类似，结构如下：

```
for (初始化; 条件; 递增) begin
  时序语句
end
```

上述结构和下面的 while 循环类似

```
初始化;
while (条件) begin
时序语句
…
递增;
end
```

代码第 10 行展现了 32 位整数在 Verilog 语言里，被直接赋值给 3 位组合输入信号，而不需要格式转换的方法。

时序仿真也被包含在图 8.11 里；在 800ns 的仿真周期里，每一个 3 位输入值

都对应 8 位输出的其中一位。在仿真的最后 200ns 内，使能输入端先被设为3'b000，随后变为 3'b011，目的是让输出端 Y 上面所有的值变为逻辑 1，最后将解码器关闭。

```
1    `timescale 1 ns/ 1 ns
2    module test_ttl138;
3    reg A, B, C, G1, G2A, G2B;
4    wire [7:0] Y;
5    initial
6    begin : gen_tests
7         integer t;
8         {G1, G2A, G2B} = 3'b100;
9         for (t = 0; t <= 7; t = t + 1) begin
10             {A, B, C} = t;
11             #100;
12        end
13        //将解码器禁用
14        {G1, G2A, G2B} = 3'b000;
15        #100;
16        {G1, G2A, G2B} = 3'b011;
17        #100;
18        $stop;
19   end
20   ttl138 uut(.A(A),
21             .B(B),
22             .C(C),
23             .G1(G1),
24             .G2A(G2A),
25             .G2B(G2B),
26             .Y(Y));
27   endmodule
```

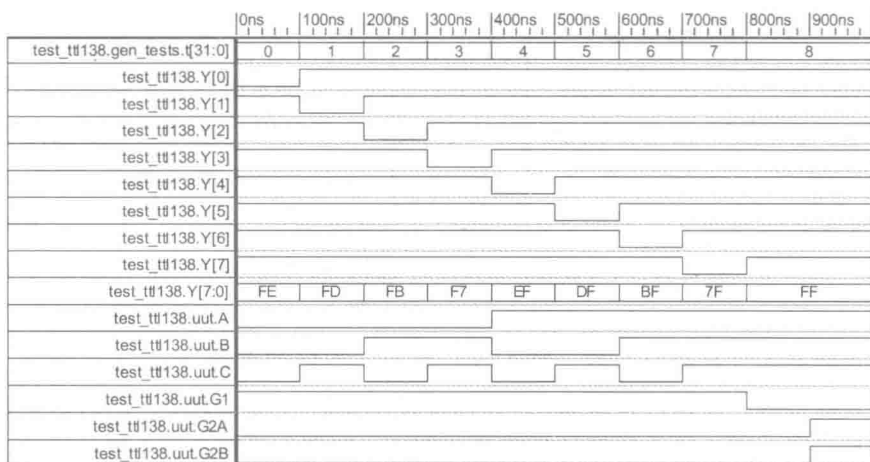


图 8.11　3－8 译码器的测试固件和仿真结果

最后需要说明的是，图 8.10 给出的解码器结构和描述与实际使用的 TTL 器件相比，还是有很大的差别。不过这是一个十分简单且又适合仿真和综合的行为模块。

图 8.12 是一个多数投票的模块，图中给出了 Verilog 源代码和模块框图。具体功能是通过统计一个 n 位的输入字里 0 和 1 的个数，然后在输出端 maj 得出一个逻辑 0 或者 1 的结果，如果是 0 则代表输入字中 0 的个数占大多数，否则就表示 1 的个数占大多数。很明显，为了让统计结果有意义，这种模块需要的输入信号的位数必须是奇数，即大于等于 3 位。

```
1     // n比特多数投票器，(n必须为奇数，且n≥3)
2     module majn #(parameter n = 5)
3                  (input [n-1:0] A, output maj);

4     integer num_ones, bit;

5     reg is_x;

6     always @(A)
7     begin
8        is_x = 1'b0;
9        num_ones = 0;
10       for (bit = 0; bit < n; bit = bit + 1) begin
11          if ((A[bit] === 1'bx)||(A[bit] === 1'bz))
12             is_x = 1'b1;
13          else if (A[bit] == 1'b1)
14             num_ones = num_ones + 1;
15       end
16    end

17    assign maj = (is_x == 1'b1)? 1'bx :
18                 (n - num_ones) < num_ones;

19    endmodule
```

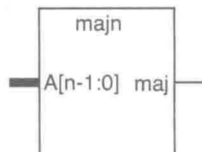


图 8.12　n 位多数投票模块的 Verilog 描述代码和框图

模块的开头（代码第 2 行、第 3 行）带有一个局部参数（parameter）n，用来定义输入的位数，设定默认值为 5。在这里使用参数（parameter）的目的是让模块用途更加广泛，因为输入信号的位数是可调的，即用户只要修改参数的设定就可以获得想要的模块。

代码第 4 行定义了两个寄存器类型的对象，但是以整数（integer）的形式体现。第一个 num_ ones 是用来统计输入信号 A 里所含的逻辑 1 的个数，而第二个 bit 是用作 for 结构的循环对象（代码第 10 到第 15 行）。还有一个寄存器变量 is_ x，是用来指示输入信号中任何未知态或者高阻态。

统计多数模块的行为描述是用 always 时序模块来完成的，从代码第 6 行开始。时序模块通过输入信号端 A 的变化来触发，并且一开始将 is_ x 和 num_ ones 初始化到全 0。随后 for 循环开始扫描输入信号的每一位，首先检查数据内部有没有未知态或者高阻态的存在，然后每检测到一个逻辑 1，就将变量 num_ ones 的数值加 1。注意一下代码第 11 行运用全等于（= = =）操作符，将输入 A 的某一位与 1'bz 和 1'bx 分别来进行比较：(A [bit] = = =1'bx) | | (A [bit] = = =1'bz)。

for 循环在代码第 15 行结束，时序模块在这里暂停并等待输入端 A 的值发生新的变化。

基于 always 模块的特性，输出信号 maj 会被持续赋值。代码第 17、18 行定义了 maj 的输出，条件语句反映出如果输入信号里存在未知态和高阻态，则将 1'bx 赋值给 maj。如果不存在未知态或者高阻态，输出信号则由输入 A 里逻辑 1 的个数和 A 的总位数的比较结果决定：(n - num_ ones) < num_ ones。

这里留给读者去验证上述语句的结果，是逻辑 1 还是逻辑 0，即输入信号 A 里的“1”的个数是不是比“0”多。

图 8.13 仿真了一个 7 位多数投票的模块。从第 5 行开始就是模块（$n=7$）的建模。initial 模块从第 6 行开始，在第 8 行将所有输入初始化为 0，随后通过用 repeat 语句向系统输入信号（代码第 9 行到第 12 行）。语句 1 < <7 是用来设定 repeat 循环执行的次数，通过把 1 向左边位移 7 次，立刻得到 2 的 7 次方。这种方法和用计算某个数的多少次方的操作符“ * * ”效果是一样的，但是并不是所有的综合软件和仿真工具都支持。

当所有可能的有效输入信号都通过模块之后，测试代码将把两个含有未知态和高阻态的值也导入模块（代码第 14 行到第 17 行），用于验证系统是否可以检测到带有 x 或者 z 的输入信号。

图 8.13 代码下方的仿真结果显示，当输入信号中逻辑 1 的数量大于等于 4 个时，模块输出信号 maj 的输出为逻辑 1。模块内部信号 num_ ones 和 is_ x 的行为，也可以通过仿真观察到。

```
1  `timescale 1 ns/ 1 ns
2  module test_majn;

3  reg [6:0] Ain;
4  wire M;

5  majn #(.n(7)) maj7(.A(Ain), .maj(M));

6  initial
7  begin
8    Ain = 0;
9    repeat (1 << 7) begin
10     #100;
11     Ain = Ain + 1;
12   end
13   #100;
14   Ain = 7'b1001x01;
15   #100;
16   Ain = 7'b000zz11;
17   #100;
18   $stop;
19 end
20 endmodule
```

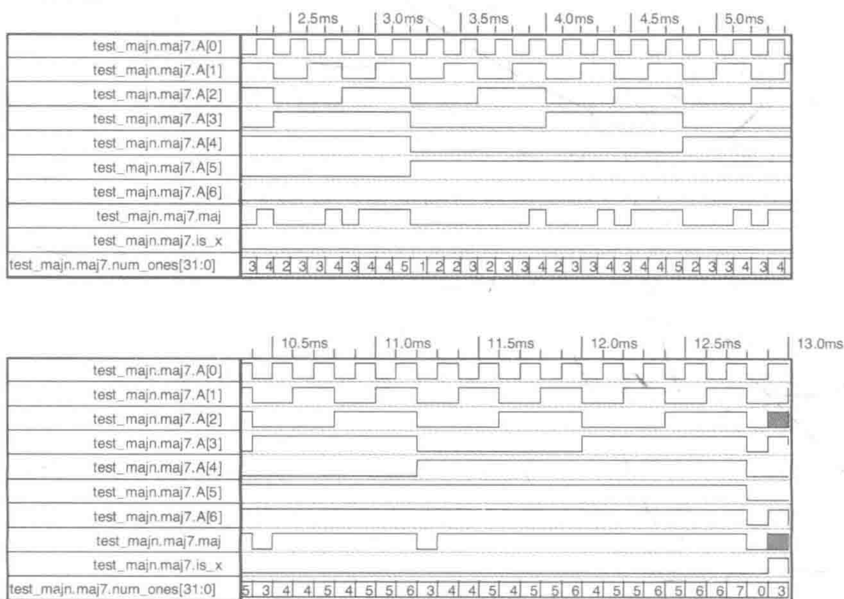


图 8.13　n 位多数投票模块测试代码和仿真结果

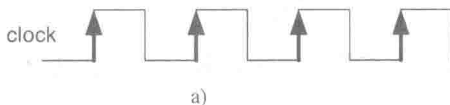
8.5 用时序模块描述时序逻辑

除了之前图 8.1 里的电平敏感锁存结构，Verilog HDL 对于时序逻辑的描述一般只用 always 模块来表达。括号中条件表述部分用预留的 posedge（上升沿）和

negedge（下降沿）两个限定词，定义了时序模块对于时钟信号的敏感度。图 8.14 是对于同步时序逻辑，使用 always 模块的一般形式，即系统中所有信号的值，要么在全局时钟的上升沿，要么在下降沿变化。

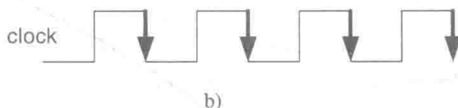
对于同一个条件表述的括号内，posedge 和 negedge 的触发机制可以同时出现；然而，这并不意味着系统是双边沿时钟触发的。上述提到的两个限定词，也可以用来描述带有异步建模机制的同步时序逻辑，在后续章节会详细介绍。

```
1 always @(posedge clock)
2 begin
3    //时序语句1
4    //时序语句2
5    ...
6 end
```



a)

```
1 always @(negedge clock)
2 begin
3    //时序语句1
4    //时序语句2
5    ...
6 end
```



b)

图 8.14　描述同步时序逻辑时 always 模块的常规形式
a）上升沿触发时序逻辑　b）下降沿触发时序逻辑

图 8.15 包含了一段代码和一个框图，描述的可能是当下最简单的时序逻辑：上升沿触发的 D 触发器。

```
1 module dff(output reg Q, input D, CLK);

2 always @(posedge CLK)
3   Q <= D;

4 endmodule
```



图 8.15　上升沿触发的 D 触发器

代码第 1 行声明了输出 Q 是寄存器型（reg），表明它必须在两个时钟的上升沿

之间保持上一次的赋值。对于关键词 reg 的使用不仅很重要，而且在这里很是必要，因为 Q 代表着一位（1bit）寄存器的状态。

第 2 行和第 3 行的 always 时序模块，只含有一行时序语句（所以不需要 begin…end 来引导），并且是非阻塞语句，表达的意思是在每一次 CLK 信号的上升沿到来时，将输入信号 D 的值赋给输出信号 Q。那么图 8.15 里的代码起始所描述的就是一个典型理想的触发器模型：与真正的硬件电路相比，它没有传输延迟，同时也没有任何数据预设和数据保持时间需要考虑。如果加入这些时序方面的元素，将会把模块复杂化，对于综合工具来说没有必要。

正如之前所说，一般建议使用非阻塞语句来描述时序逻辑。然而，值得注意的是这里对于触发器的描述，在第 3 行如果用阻塞语句的话，执行的效果是一样的。这是因为 always 模块里只有一条赋值语句。

图 8.16 是 D 触发器的测试代码和对应的测试波形。测试代码运用两个 initial

```
1  `timescale 1 ns/ 1 ns
2  module test_dff();

3  reg CLK, D;

4  wire Q;

5  initial
6  begin
7     D = 1'b0;
8     repeat (3) @(negedge CLK);
9     D = 1'b1;
10 end

11 initial
12 begin
13    CLK = 1'b0;
14    #100;
15    repeat(4) begin
16          #50 CLK = 1'b1;
17          #50 CLK = 1'b0;
18    end
19    $stop;
20 end

21 dff dut(.Q(Q), .D(D), .CLK(CLK));

22 endmodule
```

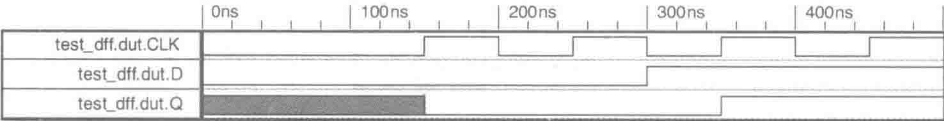


图 8.16 D 触发器测试模块和波形

时序模块产生输入信号 D 和 CLK。代码第 8 行描述了如何在测试模块里使用@(event_expression)（条件表述）；这里的 repeat 循环等待信号 CLK 上，出现 3 个连续的下降沿之后，才会将信号 D 赋值为逻辑 1。

代码下方的波形图里对应的输出信号 Q，在时钟信号第一个上升沿到来（0 到 1 的变化）之前，一直处于未知态（图中灰色部分）；换一种说法，触发器是被时钟信号同步初始化的。除此之外，尽管 repeat 循环中定义了 3 个周期后，输入信号 D 的值才会出现变化，但是在时钟的第 2 个下降沿 D 的值就被拉高了。出现这种明显的矛盾的原因，是在于 initial 模块里 CLK 信号本身的变化，即在 0 时刻，CLK 信号的值从 1'bx 变化到 1'b0，在仿真器看来，这种在最初始阶段出现的变化也属于一个下降沿。最终，可以看到输入端 D 在第 2 个下降沿被拉低后，紧接着的一个上升沿到来时输出 Q 被拉高。

下面再介绍几个用 always 模块描述一些常见的时序逻辑的例子。

图 8.17 描述的是一个带有高有效异步复位输入的 4 位二进制计数器。复位信号 reset 比时钟信号 clock 有更高的优先级，当它被激活，可以立刻强制计数器输出清零。实现这种功能的方法，是在代码第 4 行括号里条件表述部分加上 posedge reset，并在代码第 6 行到第 9 行的 if…else 构架里体现。

```
1  //带有异步复位功能的4位递增计数器
2  module cntr4(input clock, reset,
3               output reg [3:0] count);

4  always @(posedge reset or posedge clock)
5  begin
6    if (reset == 1'b1)
7      count <= 4'b0000;
8    else //同步部分
9      count <= count + 1;
10 end

11 endmodule
```

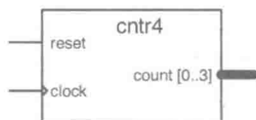


图 8.17　4 位计数器 Verilog 模块和源代码

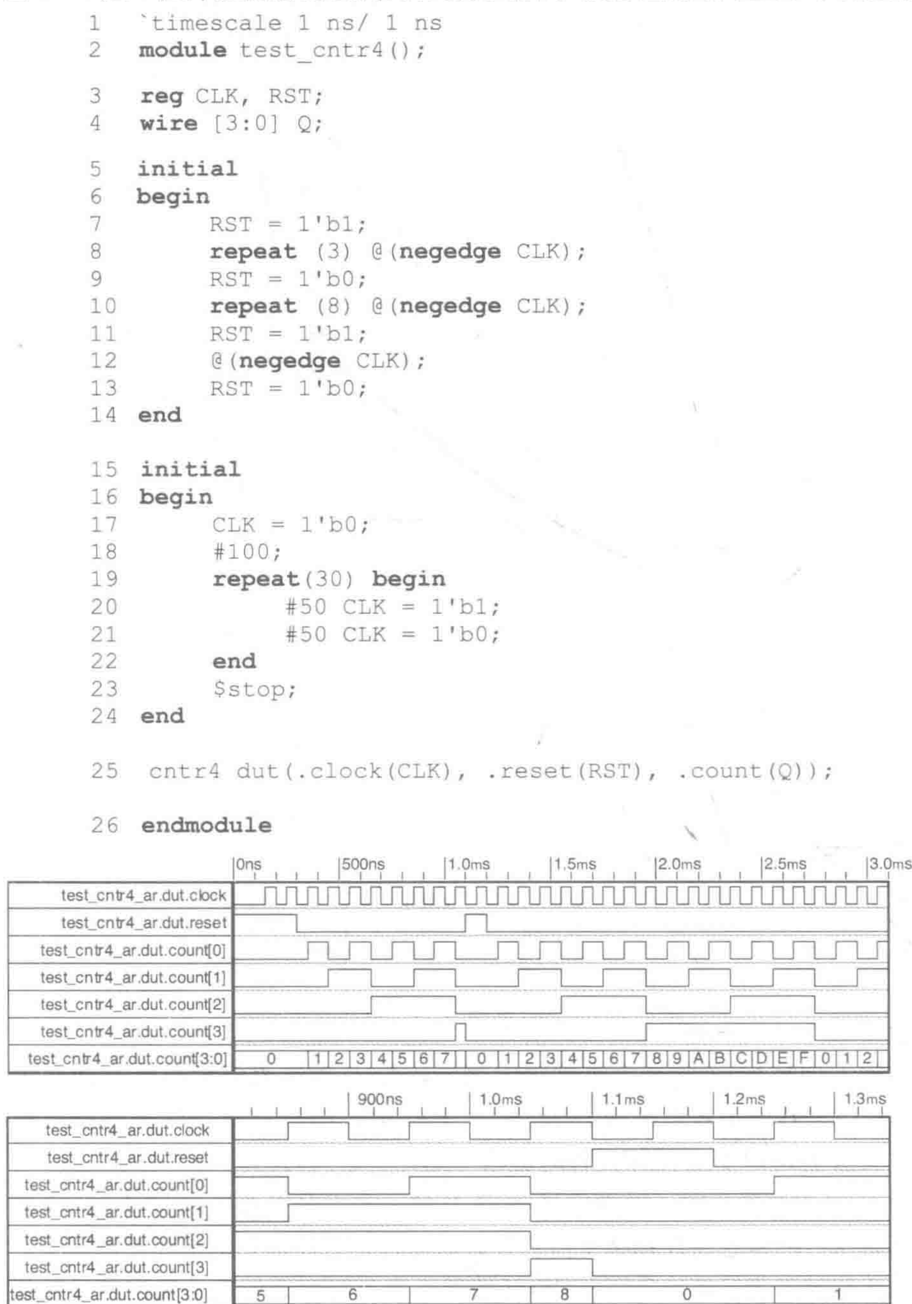
在输入信号 reset 前加上事件限定词 posedge，意味着模块可能有两套时钟逻辑。不过，当代码第 6 行出现 reset == 1'b1 时，结合前面条件表述括号里的条件，表明这里 reset 只是用来复位时钟信号的异步输入而已。

当 reset 为逻辑 0 时，时钟信号 clock 任何一个上升沿都将触发 always 模块的执行，具体执行的命令在代码第 9 行，计数器将根据时序语句的描述逐步递增。

与之前时序逻辑模块一样，4 位计数器运用非阻塞语句来给输出信号赋值，通过代码第 3 行的输出信号格式定义就能看出来，它被定义为寄存器型（reg）。大家

还可以发现 Verilog 是允许输出信号（例如这里的 count）出现在操作符的任意一边，即允许向输出信号赋值的同时，也可以读取其当前的赋值，又如代码第 9 行，把当前 count 的值递增之后赋给其本身。

图 8.18 是 4 位计数器的测试代码和仿真结果。波形图清晰地显示了计数器的



```
1   `timescale 1 ns/ 1 ns
2   module test_cntr4();
3   reg CLK, RST;
4   wire [3:0] Q;
5   initial
6   begin
7        RST = 1'b1;
8        repeat (3) @(negedge CLK);
9        RST = 1'b0;
10       repeat (8) @(negedge CLK);
11       RST = 1'b1;
12       @(negedge CLK);
13       RST = 1'b0;
14  end
15  initial
16  begin
17       CLK = 1'b0;
18       #100;
19       repeat(30) begin
20            #50 CLK = 1'b1;
21            #50 CLK = 1'b0;
22       end
23       $stop;
24  end
25   cntr4 dut(.clock(CLK), .reset(RST), .count(Q));
26  endmodule
```


图 8.18　4 位计数器的 Verilog 测试代码和仿真结果

值，在每一个时钟上升沿到来时递增一个数值，当 count 的值为十进制 8 时，复位信号 RST 被激活，计数器立刻被复位。

```
1  //带有异步低有效复位和移位使能信号
2  //的4位移位寄存器
3  module shift4(input clock, clrbar, shift, serial,
4                output reg [3:0] q);

5  always @(negedge clrbar or posedge clock)
6  begin
7    if (clrbar == 1'b0)
8      q <= 4'b0;
9    else if (shift == 1'b1) //同步部分
10     q <= {q[2:0], serial};
11 end

12 endmodule
```

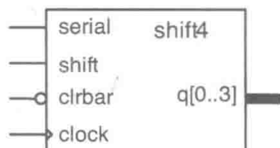


图 8.19　描述 4 位移位寄存器

与最初的设计目的一样，4 位计数器在达到最大值（4'b1111）之后，在下一个时钟上升沿到来时，自动返回到全零状态。

下一个例子如图 8.19 所示，用 Verilog 语言来描述一个 4 位移位寄存器。模块输入输出定义部分，带有一个低有效的异步复位信号 clrbar 和一个同步控制输入信号 shift，信号 shift 属于使能信号，当时钟的有效沿到来时，输出信号（这里是 4 位 reg 型）q 里的数值开始左移。

always 时序模块的触发机制在代码第 5 行：`always@ (negedge clrbar or posedge clock)`

其中限定词 `negedge` 表示输入信号 `clrbar` 的值，从逻辑 `1` 变为逻辑 `0`（下降沿）时才会触发模块的执行，配合第 `7` 行 `if…else` 构架，判断 `clrbar` 是否等于逻辑 `0` 的条件语句，在这里实现异步低有效初始化功能。

在代码第 `9` 行，程序在每次时钟上升沿到来时，判断 `shift` 的值是否为逻辑 `1`。如果条件成立，则后面的语句会将输出信号 `q` 的值更新：`q <= {q [2:0], serial};`

上述语句所做的事情，是将 q 的低 3 位数据放到高 3 位上，同时将串行数据输入 serial 的值放在 q 的最低位（bit [0]）。换句话说，当 shift 信号被激活的情况下，每来一个时钟的上升沿，q 就会执行一次 1 比特数据的左移动作。

图 8.20 是对应的测试代码和仿真结果。这里 test_ shift4 模块和图 8.16 里的 4 位计数器类似，使用了两个 initial 时序模块，一个提供仿真输入信号，另一个设定时钟脉冲。随后的仿真结果也比较直观。

```
1   `timescale 1 ns/ 1 ns
2   module test_shift4();
3   reg CLK, CLRB, SFT, SER;
4   wire [3:0] Q;
5   initial
6   begin
7         CLRB = 1'b0;
8         SFT = 1'b0;
9         SER = 1'b1;
10        repeat (2) @(negedge CLK);
11        CLRB = 1'b1;
12        repeat (3) @(negedge CLK);
13        SFT = 1'b1;
14        repeat (6) @(negedge CLK);
15        CLRB = 1'b0;
16        @(negedge CLK);
17        CLRB = 1'b1;
18        repeat (6) begin
19              @(negedge CLK);
20              SER = ~SER;
21        end
22  end
23  initial
24  begin
25        CLK = 1'b0;
26        #100;
27        repeat(30) begin
28              #50 CLK = 1'b1;
29              #50 CLK = 1'b0;
30        end
31        $stop;
32  end
33  shift4 dut(.clock(CLK), .clrbar(CLRB),
34              .shift(SFT), .serial(SER), .q(Q));
35  endmodule
```

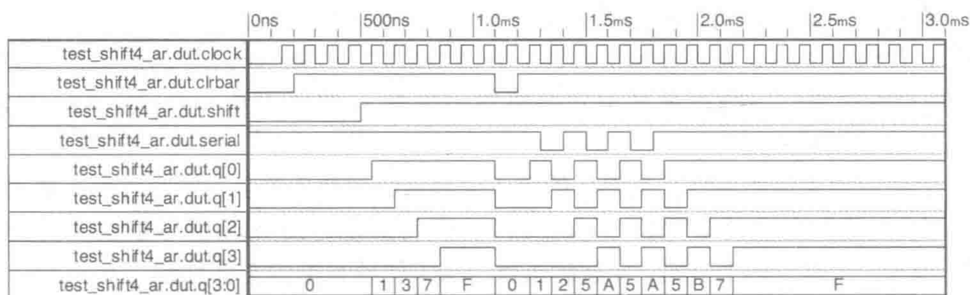


图 8.20 4 位移位寄存器的 Verilog 测试模块和仿真结果

上面 2 个例子的特点是，时序逻辑模块带有一个高有效或者低有效异步复位信号。下面会向大家介绍如何根据需要，将异步复位信号和异步使能信号植入同一个模块。

图 8.21 的代码和模块描述了一个带有正反两个输出的 D 触发器，同时模块还带有一个使能输入信号 set 和一个复位输入信号 reset，它们均为异步初始化信号，可以分别设置成高有效或者低有效。尽管这个例子只利用了高有效控制信号，但一般说来，任意高有效和低有效的组合都可以用 posedge 和 negedge 这两个限定词来描述。

```
1  //带有异步使能和复位的D触发器
2    module dff_asr(output reg q, qb,
3                   input d, clk, set, reset);

4    always @(posedge clk or posedge set
5                 or posedge reset)
6    begin
7      if (reset) begin //复位有最高优先级
8        q <= 0;
9        qb <= 1;
10     end else if (set) begin //使能的优先级排第2
11       q <= 1;
12       qb <= 0;
13     end else begin //时钟的触发的条件是使能和复位都被释放
14       q <= d;
15       qb <= ~d;
16     end
17   end
18   endmodule
```

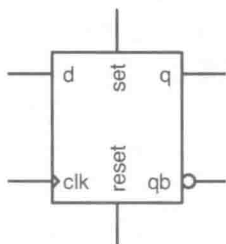


图 8.21　带有异步使能和复位信号的 D 触发器

代码第 4 行和第 5 行将 3 个输入信号，在条件表述的括号里用或（or）的关系全部列出，其中 clk 是同步时钟。随后的模块语句使用了 if…else…if…else 的构架来描述了 set、reset 和 clk 对于触发器赋值的优先级关系。注意但凡超过两条时序语句都会出现 begin…end 构架来进行引导。

有时候根据需要或者确实有必要，所有的初始化需要同步完成。这时候，所有针对寄存器型（reg）输出变量的时序语句，被全部同步到主时钟的上升沿或者下降沿上面。

图 8.22 就是针对上述要求的一个典型。其中包含一个 8 位数据寄存器的代码

和模块框图。

```
1  //带有同步复位功能的8位寄存器
2  module REG8SR(output reg [7:0] Dataout,
3           input [7:0] Datain,
4           input Rst, Clk);

5  always @(posedge Clk) //只在时钟“Clk”上升沿触发
6  begin
7    if (Rst)
8      Dataout <= 0;
9    else
10     Dataout <= Datain;
11 end
12 endmodule
```



图 8.22 模块使用同步复位的例子

代码第 5 行的条件表述中只含有输入信号 Clk 的上升沿。因此，所有关于 Dataout 的语句都跟这个条件相关，包括判断 Rst 信号是否为逻辑 1 的复位操作。

本节最后一个例子将前面介绍的一些模块里的各种特性融合在一起，其中包括可调的数据位宽、同步时钟以及行为建模等。

图 8.23 是一个被称为通用寄存器/计数器的模块，它拥有好几个不同的功能，其输入和输出的位宽是可调的。对于可调节位宽的变量，在模块定义部分，可以用 parameter（参数）来定义，这里所定义的参数名字叫 size。

作为并行数据寄存器 unireg 模块，可以用作递增或者递减计数器，还可以将数据向左或者向右进行位移。决定寄存器位宽的是代码第 2 行的参数，这里被设为默认值 8。

unireg 模块的输出信号 dataout 端口在被定义时，就表明其类型为寄存器型（reg），如代码第 6 行所示。每一个寄存器参与的动作都被同步到输入信号 clock 的上升沿上；而执行何种功能取决于一个 3 比特（3bit）位宽的控制输入信号 mode。利用信号 mode 进行功能选择的语句，是通过 case…endcase 构架来体现的（代码第 10 行到第 22 行）；信号 mode 每一种赋值都对应一个独立的分支，这里一共有 7 种操作模式，其中 mode 的值为 6 或者 7 的情况被归纳到最后的默认（default）分支里去了（代码第 21 行）。

串行数据输入分两路进入模块，一路用于左移，一路用于右移，端口分别是

```
1   //位宽可调的通用寄存器/计数器
2   module unireg #(parameter size = 8)
3              (input clock, serinl, serinr,
4              input [2:0] mode,
5              input [size-1:0] datain,
6              output reg [size-1:0] dataout,
7              output termcnt);
8   always @(posedge clock) //同步计数器
9   begin
10   case (mode)
11     0 : dataout <= 0;          //清零
12     1 : dataout <= datain;     //并行加载
13     2 : dataout <= dataout + 1;    //递增
14     3 : dataout <= dataout - 1;    //递减
15     4 : begin  //适用"<<"操作符进行左移
16       dataout <= dataout << 1;
17       dataout[0] <= serinl;
18     end
19     //运用组合操作符进行右移
20     5 : dataout <= {serinr, dataout[size-1:1]};
21     default : dataout <= dataout;  //刷新
22   endcase
23  end

24  //连续赋值语句检测数据中的逻辑0(逻辑0)
25  assign termcnt = (mode == 3) ? ~|dataout :
26                   ((mode == 2) ? &dataout : 0);

27  endmodule
```

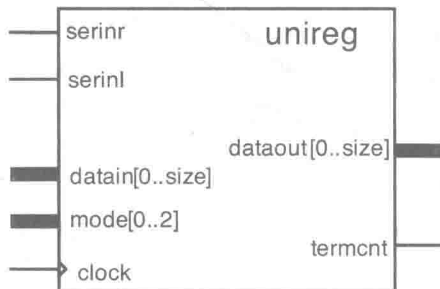


图 8.23　通用计数器/寄存器模块

serinl 和 serinr，对应到图中的代码在第 15 到第 18 行。当 mode = 4 时，模块被用作左移的移位寄存器。寄存器里的当前数据全部向左移动一位，信号 serinl 被输入到寄存器的最低位（bit [0]）。相应的操作对应代码第 16 和第 17 行的非阻塞语句。

当 mode 的值被设为 5 时，模块被用作右移的移位寄存器。代码第 20 行用另一种代码风格诠释了右移的操作，这里运用组合操作符将寄存器里的位数据向右移动一位，放在低 size - 1 位的位置。同时将 serinr 输入信号放在寄存器的最高位（最左边）。

模式 0 ~ 3 对应的功能很明显，不需要过多解释，具体读者可以阅读代码第 11 行到第 14 行。

剩下的操作模式都用 case 语句里的 default 来表示。这是一个刷新模式，当模式的值为 6 或者 7 时都将对应 default 的操作。这里的语句就是将当前寄存器的值，直接赋给输出 dataout，即本身。这种操作还有一种更加简便的表达方法：

```
default: ; //用空语句来表示刷新
```

空语句（;）意思是“什么都不做”的语句；上述语句的意思是输出信号 dataout 寄存器由于没有新的赋值而保持自身现有的值。因此，用图中的代码或者这里介绍的空语句来描述让一个寄存器保持现有的值，完全取决于设计者的喜好。

模块 unireg 的另一个输出信号是线网型（wire）变量 termcnt，它存在的目的是当模块被用作递增或者递减计数器时，可以指计数的最大值或者最小值。

由于寄存器 dataout 的位宽是可以调节（自定义）的，因此很难用它和一些固定的最大值来进行比较，例如 8'hFF。这个问题可以在语句中，用条件操作符结合按位逻辑操作符解决，图中代码第 25 行和第 26 行的连续赋值语句给出了答案：

```
assign termcnt = (mode = = 3)? ~ |dataout: ((mode = = 2)?
&dataout: 0);
```

上述语句判断操作模式到底是“递增”（2）计数还是“递减”（3）计数，如果是递增，则让输出信号 dataout 进行按位与操作，并把结果赋予 termcnt；如果是递减，则让输出信号 dataout 进行按位或非，并把结果赋予 termcnt。因此可以推断如果操作模式是 2 或者 3，则 termcnt 的最终结果都会是逻辑 1，否则它最终结果将会是逻辑 0。

图 8.24 是测试模块 Test _ unireg 的代码，用来运行通用寄存器/计数器模块的仿真。代码第 3 行的局部变量（test _ size）被赋予固定的值，并影响到整个模块。这里，test _ size 的值是 4。这个设定影响到并行数据输入 datain 和数据输出 dataout 的位宽（代码第 7 行到第 9 行），以及通用寄存器/计数器模块的位宽（代码第 12 行）。

测试代码含有两个 initial 时序语句模块，第一个是（代码第 20 行到第 25 行）用来产生的时钟信号；第二个（代码第 26 行到第 49 行）用来产生仿真流程，根据不同的模式（mode 的不同赋值），来测试通用模块的各种功能。代码的下方是仿真结果波形图。

将输入信号 mode 赋值为 0 并清空寄存器之后，延迟 200 个时钟周期，寄存器被设为递增计数器（mode = 2），且执行 30 个时钟周期的计数。在仿真波形图中，可以很明显地看到输出数据在以二进制的格式递增，在此过程中当输出数据中的每一位都变为逻辑 1 时，终止计数（termcnt）信号被拉高，之后计数器开始新的一轮计数。

随后模式被切换为递减计数器（mode = 3），持续时间仍然是 30 个时钟周期。此时输出数据开始逐步递减，在输出信号的每一位都变为逻辑 0 时，终止计数信号再次被拉高。紧接着程序开始测试模块的其他功能，左移（mode = 4）、右移（mode = 5）、并行加载数据（mode = 1）和数据刷新（mode = 7）等（代码第 38 行到第 47 行）。整个仿真在代码第 48 行全部结束。

```
1   `timescale 1 ns/1 ns
2   module Test_unireg();
3   localparam test_size = 4; // unireg位宽
4   // 输入
5   reg clock, serinl, serinr;
6   reg [2:0] mode;
7   reg [test_size-1:0] datain;
8   // 输出
9   wire [test_size-1:0] dataout;
10  wire termcnt;
11  // unireg 模块建模，位宽为4bit
12  unireg #(.size(test_size))
13              mut(.clock(clock),
14              .serinl(serinl),
15              .serinr(serinr),
16              .mode(mode),
17              .datain(datain),
18              .dataout(dataout),
19              .termcnt(termcnt));
20  initial // 产生周期为100ns的时钟
21  begin
22        clock = 0;
23        forever
24              #50 clock = ~clock;
25  end
26  initial  // 导入测试输入信号
27  begin
28        serinl = 0;
29        serinr = 1;
30        mode = 0;
31        datain = 'h9;
32        #200 mode = 2;
33        repeat (30)  // 等待30个时钟周期
34              @(posedge clock);
35        mode = 3;
36        repeat (30)
37                @(posedge clock);
38          mode = 4;
39          repeat (8)
40                @(posedge clock);
41          mode = 5;
42          repeat (8)
43                @(posedge clock);
44          mode = 1;
45          #400 mode = 2;
46          #800 mode = 7;
47         #1000;
48         $stop;
49   end
50  endmodule
```

图 8.24 通用寄存器/计数器仿真程序和波形图

图 8.24　通用寄存器/计数器仿真程序和波形图（续）

8.6　描述存储芯片

本节将介绍一些非常简单的小模块，它们可以被看作是 RAM 和 ROM 一些最基本的仿真模型。与存储芯片供应商相比，虽然这些模块在时序上不够精准，功能仿真也不够成熟，但是当需要快速构建一些带有存储功能的模块，并能够参与大型系统仿真时，这些模块是可以有效胜任的。

本节涉及一些前面章节已经介绍过的 Verilog 编程描述方面的内容，例如位宽的变化、参数的使用，以及用时序语句来构建行为模块等。除了这些重要的特性之外，在设计存储模块的同时，本节还会向大家介绍一些前面章节没有提到的特性例如：

1）数组——构建一个存储单元时所要用到的基本构架；

2）双向端口——一个既可以输入也可以输出的端口；

3）存储单元初始化——用一个已有的文件向存储单元加载数据。

Verilog 语言不支持新建一个复合类型的变量，例如数组或者记录。而一组寄存器型（reg）变量，可以用如下语法格式来定义（线网型变量的定义方式也类似）：

//一组寄存器变量的个数是 *m*，每个都是 *n* 比特

```
reg [n-1 : 0] mem [0 : m-1];
```

上述语句表示一组变量有 m 个元素，每一个寄存器变量的位数是 n bit。通过这种方法，对象 mem 可以看作是一个二维的数组变量，即一个内存单元。

随着 Verilog-2001 标准的发布，Verilog 语言在关于数组变量的处理上得到了有效增强，其中最明显的两个改进的地方，是可以识别多维数组变量以及处理数组中某一位数据的能力。虽然上述改进在本处存储模块中用不到，但是有兴趣的读者可以阅读参考文献 [2]。

另外一个存储模块中，比较常见的特性是双向数据交换。大部分 RAM 使用双向三态数据总线来进行数据的读写，信息交换和访问都是通过同一个总线进行。Verilog 语言里用 inout 端口来表达双向模式，同时内置仿真支持高阻态和多路信号的驱动。用户必须注意 inout 端口是线网型（wire），驱动可以是单个，也可以是多个。例如在读数据过程中，inout 端口是被存储单元中的数据驱动的，否则它会被设置为高阻态。在向 RAM 写数据的过程中，端口由外部数据驱动，并且存储单元会向数据总线发送高阻态，随后总线会在外部数据到来时，将高阻态更新为需要向存储单元写入的数据。

图 8.25 是一个简易 RAM 模块的框图和 Verilog 描述代码。模块是通用的，在规定的范围内提供灵活可变的地址和数据总线，形成不同规模的存储单元。

模块 ram 代码的第 4 行声明了两个参数 Awidth 和 Dwidth。它们决定地址 address和数据 data 的位宽，并对应代码第 6、7 两行的定义。代码第 5 行定义的 3 个低有效控制信号的作用分别是：

1）web——写使能信号，其值为逻辑 0 时，向存储空间写入数据；

2）ceb——芯片使能（片选）信号，允许存储空间被访问并进行读写操作；

3）oeb——输出使能信号，在读操作过程中将存储空间里的数据发送到数据端口输出。

存储空间的数据位宽和地址位宽之间是 2 的指数关系，即 2^{Awidth}。代码第 8 行的局部变量使用左移操作符来进行运算（因为不是所有的仿真器都支持“ * * ”操作符）。

随后局部变量 Length 在代码第 9 行用来声明存储空间的数据位宽。

代码第 11 行和第 12 行用连续赋值语句描述了存储空间的读操作：

```
assign data = (~ceb & ~oeb & web) ?
            mem [address] : 'bz;
```

语句操作符（ = ）右边的任意一个信号状态发生变化时，都会触发其立刻执行，这包括所有的存储控制输入信号和地址。

在读操作中“写使能”信号必须为逻辑 1，这个条件可以有效防止所谓总线冲突的现象，这种冲突的具体表现是模块同时进行读操作和写操作。

读取的数据在被访问时，用类似 C 语言的数组索引符号（[ ]），在访问单比特

```
1    //静态随机存储器访问模块
2    //是地址线宽度
3    //是数据线宽度

4   module ram #(parameter Awidth = 8, Dwidth = 8)
5            (input web, oeb, ceb,
6            inout [Dwidth-1:0] data,
7            input [Awidth-1:0] address);

8   localparam Length = (1 << Awidth);

9   reg [Dwidth-1:0] mem[0:Length-1]; //存储器阵列

10  //存储器读操作
11  assign data = (~ceb & ~oeb & web) ?
12           mem[address] : 'bz;

13  //存储器写操作
14  always @(posedge web) //在web信号的上升沿到来时触发
15    if ((ceb == 1'b0) && (oeb == 1'b1))
16      mem[address] = data;

17  endmodule
```

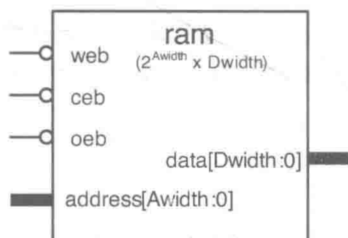


图 8.25　简易 RAM 的 Verilog 描述和框图

数据或者寄存器、线网变量的某一部分数据时，也是同样的格式。

需要指出的是，Verilog-1995 标准不允许在访问数组时，同时访问单个比特的数据，或者其中的一部分数据。因此，在标准 Verilog-2001 中，这是一项重大改进。不过 1995 的标准对此处的简易存储模块影响不大，因为所有存储空间的访问，都是以字（32 位）为单位访问。

代码第 11 行和第 12 行所用的连续赋值语句和 data 的端口定义 inout 是吻合的，此时端口的特性和线网型（wire）变量没有区别。如果条件语句中“?”之前的部分不能满足，连续赋值语句将会给双向数据端口赋值为高阻态。

存储单元写操作，由代码第 14 行到第 16 行的 always 模块执行。输入数据在低有效信号“写使能”的上升沿到来时，被锁存到存储单元中，此时还必须符合存储单元被激活，以及无法进行读操作这 2 个条件。这种情况下，双向数据端口被用作输入（线网型）；Verilog 仿真器可以自动识别连续赋值语句产生的高阻态和外部输入数据两种不同的来源。

图 8.26 是 ram 模块的 Verilog 测试代码。对于测试模块 test _ ram 来说，一个重要的问题是需要将双向数据端口定义成线网型（wire），而不是寄存器型（reg）。对于一个纯输入端口来说通常也是线网型。

```
1     //16字节RAM测试模块

2     `timescale 1 ns/ 1 ns
3     module  test_ram;

4     reg webar, oebar, csbar;
5     reg [7:0] datareg;
6     reg tri_cntr;     //数据高阻使能控制信号
7     reg [3:0] address;
8     // 数据输入/输出三态缓冲
9    wire [7:0] data = (tri_cntr == 1'b1)?
10                              datareg : 8'bz;
11   initial
12   begin : test
13        tri_cntr = 1'b1;     //将数据准备好
14        webar = 1'b1; oebar = 1'b1;
15        csbar = 1'b1; datareg = 8'b01010101;
16        address = 4'd0;
17        #10 csbar = 1'b0;
18        repeat (10) //10个写操作
19        begin
20             #10 webar = 1'b0;
21             #10 webar = 1'b1;
22             #10 address = address + 1;
23             datareg = ~datareg;
24        end
25        address = 4'd0;
26        tri_cntr = 1'b0;  //数据输出高阻
27        repeat (10)  //10个读操作
28        begin
29             #10 oebar = 1'b0;
30             #10 oebar = 1'b1;
31             #10 address = address + 1;
32        end
33        $stop;
34   end
35   ram #(.Awidth(4), .Dwidth(8))
36             ram_ut(.web(webar),
37                  .oeb(oebar), .ceb(csbar),
38                  .data(data), .address(address));
39   endmodule
```

图 8.26　RAM 模块测试代码和仿真结果

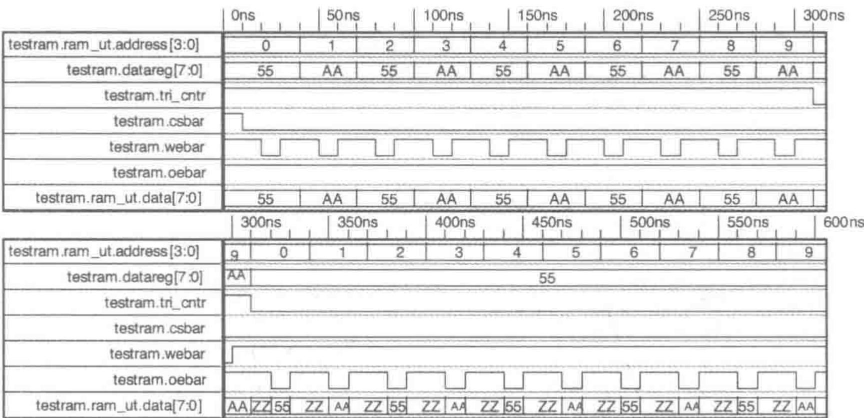


图 8.26　RAM 模块测试代码和仿真结果（续）

线网型变量 data 在测试代码第 9 行和第 10 行被连续赋值语句驱动，在模块进行读操作时必须被赋予高阻态。

为了实现上述功能，测试模块里用到一个单比特寄存器变量 tri _ cntr（三态控制的简写），目的是为了控制写操作。在写操作过程中，tri _ cntr 被设为逻辑 1，使得 datareg 可以被写入存储单元。而在读操作过程中，tri _ cntr 被设为逻辑 0，这样 data 总线会被赋予高阻态。

代码第 35 行到第 38 行，定义了一个 16 字节的 ram，方法就是分别修改地址和数据位宽的局部变量为 4 和 8。代码第 11 行开始的 initial 时序模块，首先进行 10 次写操作，将数据写入地址空间 0 ~9；10 个地址空间被交替写入十六进制的8'h55 和 8'hAA。整个写操作完成后，测试程序让地址空间复位，并将 tri _ cntr 拉低，数据总线随后被赋予高阻态（代码第 26 行）。第二个 repeat 循环（第 27 行到第 32 行）进行 10 次读操作，顺序也是从地址空间的 0 ~9 依次执行。

图 8.27 给出了一个上述过程所对应的功能框图。

仿真结果在图 8.26 的代码下方。可以看到每次 webar 信号的脉冲，都对准了相应的地址空间和数据。通过拉低信号 tri _ cntr，模块切断了外部数据输入，并开始进行读操作，每次读操作都对应信号 oebar 的脉冲和地址计数的递增。

当需要存储和读取固定数据时，可以用 ROM 模块。例如可以将测试程序和测试数据存在 ROM 里面，用来和被测模块进行对接，执行整个测试过程。

图 8.28 给出了一个简易 ROM 的 Verilog 代码和框图。和上述 RAM 一样，存储模块被设计成大小可调，用局部变量定义地址和数据的位宽。

有了之前 RAM 的例子作为参考，对于 ROM 模块来说，它使用一个 localparam（局部变量）根据地址的位宽去计算存储单元的大小（代码第 6 行），随后在下一行定义存储单元（代码第 7 行）。模块的行为在代码第 8 行用连续赋值语句来描述；具体体现为在输出使能信号 oeb 被激活的情况下，将当前地址空间 address 对

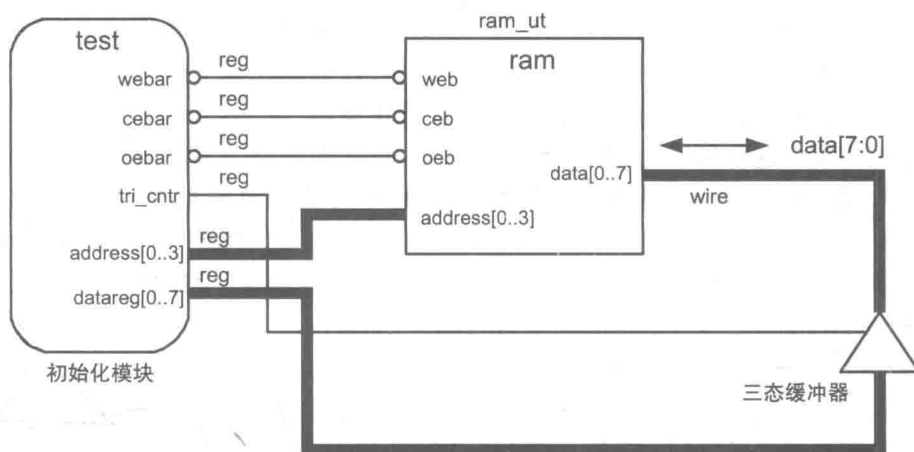


图 8.27　test _ ram 模块功能框图

```
1  //可变位宽只读存储器
2  module rom #(parameter Awidth = 8, Dwidth = 8)
3              (input oeb,
4              output [Dwidth-1:0] data,
5              input [Awidth-1:0] address);

6  localparam Length = (1 << Awidth);

7  reg [Dwidth-1:0] mem[0:Length-1]; //存储器阵列

8  assign data = (oeb == 1'b0) ? mem[address] : 'bz;
9  endmodule
```

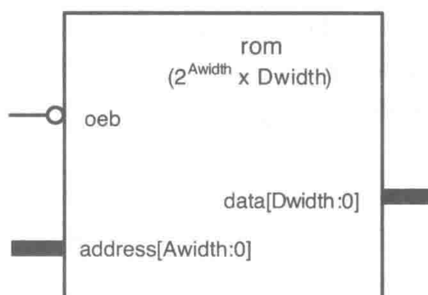


图 8.28　ROM 模块的 Verilog 语言描述

应的存储单元 mem 的内容，赋给输出端口 data。注意在 ROM 模块里，输出端口的格式是 output 而不是双向的 inout，因为对于 ROM 来说只有读操作。当输出使能信号被置高时，输出端口会被设为高阻态。

ROM模块里存储的实际数据并没有在Verilog代码里有具体的描述。这种情况下进行仿真的话，需要从外部引入数据，可以将数据保存在一个ASCII文本文件里，然后在开始仿真之前将数据加载到ROM模块里。

根据需要，这种初始化ROM的方式对于RAM模块也适用。而且这也给第三方工具（例如编程器），提供一个较为方便的将大批量数据加载到存储单元的途径。

对于通过文本文件将数据加载到存储空间的“系统命令”有两种：

- `$readmemb ("filename", array_name);`
- `$readmemh ("filename", array_name);`

这两个命令的区别在于保存在被加载文件中的数据格式：第一条命令需要输入文件的数据格式是二进制，而第二条命令需要数据格式为十六进制。

代码8.2给出了一个数据格式的模板，其中所有被加载的数据都是二进制。第一行“@”后面的数字，代表数据即将被写入的地址空间，它是十六进制的。通常来说写数据都是从地址0开始。这里所用到的分隔符@hex_address格式，可以让存储空间在初始化时，对应不同地址空间和不同的数据。

```
@0
1010 0000 1111 1011 0010 1001 0110 1110
0111 1101 1011 1111 0000 0001 0010 0101
1010 0000 1111 1011 0010 1001 0110 1110
0111 1101 1011 1111 0000 0001 0010 0101
```

代码8.2 文件rom_data.txt的内容

实际的数据内容以4位二进制数为一组，用空格或者换行符分开，并按照被存入存储单元的顺序排列。如果文件中包含的数据小于存储单元的容量，那么剩余的存储单元将保持空闲的状态。

命令中文件名“filename”对应的位置，是填写被加载的文件的真实名称。对于文件名的格式，取决于执行Verilog仿真的操作系统。不过通常来说，只要加载的文件的保存路径和测试代码文件相同，引号内所需填写的内容就只是一个文件名而已。

对于系统命令$readmemb（）和$readmemh（）的调用，可能包含在存储单元的测试代码中，此时，命令里的array_name所对应的位置就是存储单元本身，例如图8.28中的mem。

ROM存储单元的初始化是在测试模块Test_rom的代码里完成的，如图8.29所示。代码第6行和第7行用一个initial模块语句，将代码8.2里的数据加载到存储单元里：

```
$readmemb ("rom_data.txt", dut.mem);
```

上述语句里的dut.mem必须在实例名rom及其定义之前出现，实例rom的定义

```
1     `timescale 1 ns/ 1 ns
2     module Test_rom();

3     wire [3:0] Data;

4     reg [4:0] Address;

5     reg oebar;

6     initial //加载文件中的数据并将ROM初始化
7       $readmemb("rom_data.txt", dut.mem);


8     rom  #(.Awidth(5), .Dwidth(4))
9                dut(.oeb(oebar),
10               .data(Data),
11               .address(Address));

12    initial
13    begin
14      Address = 0;
15      repeat (32) //读取整个ROM的内容
16      begin
17        oebar = 1'b1;
18        #25 oebar = 1'b0;
19        #50 oebar = 1'b1;
20        #25;
21        Address = Address + 1;
22      end
23      $stop;
24    end
25   endmodule
```

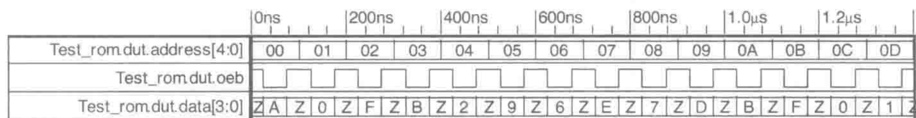


图 8.29　ROM 所对应的 Verilog 测试模块

在代码第 8 行到第 11 行。ROM 里默认的地址和数据的位宽被加载的文件覆盖，本例中，存储单元被定义为“32 ×4”（32 个字，每个字 4 比特）；这个和 rom _ data. txt 文件中的格式是一样的。

剩余的测试代码（第 12 行到第 24 行）是一个 initial 模块语句，所做的事就是把所有地址空间对应的数据读出来，从 0 ~ 31。代码下面的波形图也验证了这一过程；观察仿真波形图，可以发现当信号 oebar 被激活时，所读出的数据和预先输入到 rom _ data. txt 文件里的数据是一致的。

关于用 Verilog 描述存储单元的最后一个例子，就是向大家介绍另一种描述

ROM 的方法。代码 8.3 是一个用 Verilog 描述的模块，名为 rom _ case。从模块名称可以看出，它运用 Verilog 里的 case... endcase 时序语句来描述 ROM。

```
1 //用case 语句描述 ROM
2 module rom _case # (parameter Awidth =8, Dwidth =8)
3              (input oeb,
4              output [Dwidth -1 : 0] data,
5              input [Awidth -1 : 0] address);
6 reg [Dwidth -1 : 0] data _ i;

7 always @  (address)
8 begin
9  case (address) //定义 ROM 的内容
10    0: data _ i = 'h88;
11    1: data _ i = 'h55;
12    2: data _ i = 'haa;
13    3: data _ i = 'h55;
14    4: data _ i = 'hcc;
15    5: data _ i = 'hee;
16    6: data _ i = 'hff;
17    7: data _ i = 'hbb;
18    8: data _ i = 'hdd;
19    9: data _ i = 'h11;
20    10: data _ i = 'h22;
21    11: data _ i = 'h33;
22    12: data _ i = 'h44;
23    13: data _ i = 'h55;
24    14: data _ i = 'h66;
25    15: data _ i = 'h77;
26    default : data _ i = 'h0; //这里可以使用"x"或者"0"
27  endcase
28 end
29 //用三态缓冲器来输出
30 assign data =  (oeb = =1'b0)? data _ i  : 'bz;

31 endmodule
```

代码 8.3　用 Verilog 的 case 语句描述 ROM

模块一开始和图 8.28 代码里的开头是一样的，随后定义了一个寄存器变量 data _ i，位宽是 Dwidth。这个变量的作用是在三态门判断是否送出数据之前，将 case 语句上的输出暂时锁存。

always 模块只对输入地址 address 的变化有响应，随后的 case 语句会立刻将每一个数据映射到相应的地址空间。这样，存储单元里所含的“内容”就在模块中被明确了，而不需要用外部文件去加载。不过这种方法只适用于空间不大的存储单元，因为在编写代码时，每一个地址空间所对应的数据必须明确。

当总的数据量小于存储单元的容量（2^{Awidth}）时，代码第 26 行的默认语句将所有没有用到的存储空间的赋值进行定义。如果 ROM 模块用组合逻辑电路来表达的话，那么默认值里用 x 比用 0 需要更少的逻辑门，因为对于逻辑综合工具来说，x 代表着“任意值”。

8.7 描述 FSM

本节主要向大家介绍如何使用 Verilog HDL 描述 FSM 的行为。面对众多复杂的电子系统，FSM 如今已经变为系统设计师最重要的工具之一。一旦确定了状态图，由 Verilog 支撑的行为语句，可以快速并直接地建立同步 FSM 的仿真模型。结合适用范围较广，功能强大的逻辑综合工具软件，FSM 的设计如今变得十分快速和有效。

图 8.30 是一个同步 FSM 的系统框图。其中包含了两个主要模块：状态寄存器和输出/下一状态逻辑电路。两者之间用反馈信号进行相连。对于基本模型来说有几种情况；不过状态寄存器通常含有一定数量的触发器（对于 n 个触发器而言，2^n 必须大于或等于 FSM 的状态数），而输出/下一状态逻辑电路模块，包含了判断下一个状态和输出端赋值的组合逻辑电路。

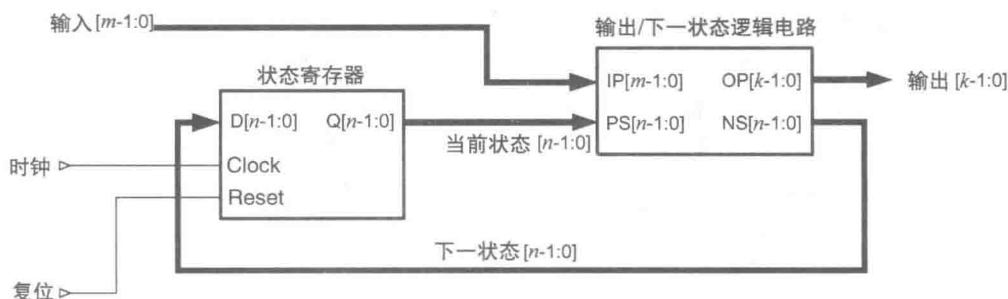


图 8.30　FSM 系统框图

图 8.30 所示的 FSM 通常被称为米利型 FSM，k 位输出受到 n 个状态和 m 位输入的影响。FSM 的初始化可以用一个异步 Reset 信号，让所有状态触发器进入已知态（通常是清零）。米利 FSM 的一个潜在问题是异步输入信号 Input 上面如果发生变化，会在输出端 Output 得到体现。为避免这种情况的发生，可以让输出的变化只取决于当前的状态（信号 Present - state），即状态寄存器的输出。这种设计调整

产生了所谓的摩尔 FSM。因此本节讲述的重点将放在如何用 Verilog 描述这两种 FSM。

设计任何 FSM 都要从绘制状态图开始。它提供了关于 FSM 行为的图像描述方式，方便设计者决定系统需要的状态数和它们之间的逻辑关系。一旦状态数量决定了，下一步就是用二进制码来标记每一个状态，这被称为“状态分配”。在 Verilog 里，状态分配的方法可以有以下几种：局部参数；模块标题部分定义的参数；用 `define编译指令定义。

第一种大概是最常见的，因为状态编码对于描述 FSM 时，似乎应该是一组固定的数值组合。下面一段 Verilog 代码诠释了如何用局部变量定义一个 4 状态的 FSM：

```
localparam s0 =2'b00,
           s1 =2'b01,
           s2 =2'b10,
           s3 =2'b11;
```

通过上述声明，可以在后续的代码中，用 s0... s3 来代替二进制码表达每一个状态，增加代码的易读性。

在模块标题后面的括号里，定义状态值拥有更多的灵活性。可以在需要时，将每一个状态所对应的值重新设定，如下代码所示：

```
//在模块标题后面直接定义参数
module fsm # (parameter s0 =0,
              s1 =1,
              s2 =2,
              s3 =3)
     (input clk, ... , output ...);
//重新定义默认的参数值
fsm # (.s0 (2),
      .s1 (0),
      .s2 (3),
      .s3 (1))
     F1 ( .clk (CLK), ...);
```

第三种方法利用`define 编译指令对状态进行定义，它和在 C/C + + 里用#defi进行文本替换的原理是一样的。编译指令必须在模块标题之前定义完成，如所示：

```
`define WAIT 4'b001
`define IDLE 4'b011
`define ACK1 4'b101
```

```
`define ACK2 4'b110

module fsm (...);
```

在上面 FSM 模块 fsm 的内部，引用某个状态值的格式如下所示：

```
//在标识前必须加上着重符
Present - State  < = `IDLE;
```

图 8.30 里的状态寄存器模块是通过一个 always 时序模块进行描述的。因此，所有输出端口必须定义成寄存器型，如下所示：

```
reg [n - 1  : 0] Present - State; //状态的数量必须为 2^n
```

状态寄存器对应的时序模块代码如下：

```
1 always @  (posedge Clock or posedge Reset)
2    begin
3    if (Reset = = 1'b1)
4       Present - State  < = s0;
5    else
6       Present - State  < = Next - State;
7    end
```

代码 8.4　状态寄存器的 always 时序语句模块

对照之前章节的内容，代码 8.4 描述了一个带有高有效异步初始化（低有效异步初始化也适用）信号的同步时序逻辑。

当时钟信号 Clock 每次来一个脉冲（0 到 1 的变化）时，当前状态 Present - State 信号的值被下一个状态 Next - State 更新（代码第 6 行），后者是通过输出/下一状态逻辑模块输出的。因此现在当前状态 Present - State 信号变成了输出/下一状态逻辑模块的输入信号，它通过结合输入信号的变化和当前输入信号的状态，更新下一个状态 Next - State 的输出值。反馈信号 Next - State 随后继续等待下一个时钟脉冲的上升沿到来，循环更新当前状态 Present - State 的值。

将输出/下一状态逻辑模块分成两部分，也是一个很好的尝试，一部分专门处理输出信号，另一部分专门处理下一个状态。这样代码变得更加好理解，也很容易维护。代码 8.5 是一段适用于"下一状态"的 Verilog 源代码。

```
1 always @  (Present - State, Input1, Input2, Input3...)
2 begin
3 //考虑每一个可能的状态
4  case (Present - State)
5    s0: if (Input1 = = 1'b0)
6          Next - State  < = s1;
7       else
8          Next - State  < = s0;
```

```
9     s1: ...;
10    s2: ...;
11    default : Next - State  < =s0;
12  endcase
13 end
```

代码 8.5　描述下一个状态的 always 模块

这里的 always 模块使用组合逻辑描述下一状态。因此，8.2 节所讨论的准则在这里必须都要满足，这样变量 Next - State 才算被赋予了所有可能的值（代码 8.5 中的 default 语句实现了这一目的）。

always 时序模块必须对当前状态 Present - State 和 FSM 的所有其他输入信号做出响应，如代码第 1 行所示。代码第 4 行到第 12 行所对应的 case... endcase 语句，根据输入信号值的变化，考虑到了所有 Next - State 可能的情况。通过这种方法，这部分代码同时也用行为语句描述了符合状态图的状态切换过程。事实上信号 Next - State 的赋值是通过 always 时序语句模块来完成的，意思就是它必须和信号 Present - State 的声明格式是一样的：

```
reg [n -1  : 0] next - state; //组合逻辑行为输出
```

对于图 8.30 里输出/下一状态逻辑模块中驱动 FSM 输出的部分，可以用另一个 always 模块来描述，或者用连续赋值语句来完成。对于选择何种方法，取决于输出逻辑的复杂程度。对于摩尔型 FSM，输出只和当前状态相关，因此表达能力较强的连续赋值语句，一般来说比较适合。对于米利型 FSM 来说，输出逻辑更加复杂，因此需要时序模块来表达，同时不要忘记将输出信号定义为寄存器型(reg)。

下面给出用连续赋值语句，来描述一个简单的米利型 FSM 的输出的例子：

```
assign Output 1 =  ( (Present - State = = s0)
                    && (Input1 = =1'b0)) | |
                     ( (Present - State = = s2)
                    && (Input2 = =1'b1));
```

这里的 Output1 直接取决于当前态和输入信号的值。对于状态寄存器和下一个状态反馈逻辑电路，还可以将它们并到同一个 always 时序模块里，而不是分成两个模块。这样做的好处是让 Verilog 代码更加简洁，并且将代码 8.4 里时序逻辑的行为和代码 8.5 里的组合逻辑的行为进行了合并，如代码 8.6 所示：

```
1 ...
2 reg [n -1  : 0] state; //单个状态寄存器
3 ...
4 always @  (posedge clock or posedge reset)
5 begin
6 if (reset = =1)
```

```
7  state  < =State0;
8 else
9  case (state)
10    State1: if (Input1 = =0)
11       state  < =State2;
12      else
13       state _ reg  < =readlone;
14    State2: if (Input2 = =1)
15       state < =State3;
16      else
17       state < =State2;
18 ...
19    default : state < =3'bxxx;
20   endcase
21 end
```

代码 8.6 将状态寄存器和下一状态逻辑合并到同一个 always 模块

通过将状态寄存器和下一状态逻辑组合成一个时序语句模块，还可以达到少用一个寄存器（reg）变量的效果，当在同一个模块里进行描述时，当前状态和下一状态就没必要分开定义了。正如代码 8.6 第 2 行所示，只有一个名为 state 的寄存器型变量需要声明。赋值的行为（第 7、11、15... 行）和读取结果（第 9 行）的行为描述，均通过这个综合的信号来完成。整个时序模块通过时钟的上升沿或者 reset输入信号的上升沿（假设这里是异步高有效初始化）触发。在测试了复位信号之后（第 6 行），整个代码的行为描述和代码 8.5 里的下一状态逻辑描述十分类似，利用 case... endcase 语句将每一个状态和输入条件都囊括进来，与最初设计时的状态图完全吻合。

事实上，代码的第 9 行到第 20 行描述的是一个自相关同步反馈逻辑系统，其中信号 state 既是一组 D 触发器的输出，也是 case 语句里 if... else 套嵌语句里对应的每一个触发器的输入。

下面介绍两个针对上述内容的实例。第一个是用 FSM 控制一个计时器，用于统计两个国际象棋棋手之间的比赛；第二个是带有自动落锁功能的密码锁。

8.7.1 实例 1：国际象棋比赛计时器

图 8.31 是一个国际象棋比赛计时器的系统框图，参加比赛的选手是两名，计时器将统计每一名棋手用来考虑下一步如何走棋所用的时间。参赛选手，即图中的选手 A 和选手 B 都各有一个自己的计时器（图中的计时器 A 和计时器 B），系统的设计目的是在比赛开始之后以小时、分钟和秒为单位统计所用的比赛时间。

对于计时器的内部运行逻辑不在本例的讨论范围内，主要关注的是如何使用

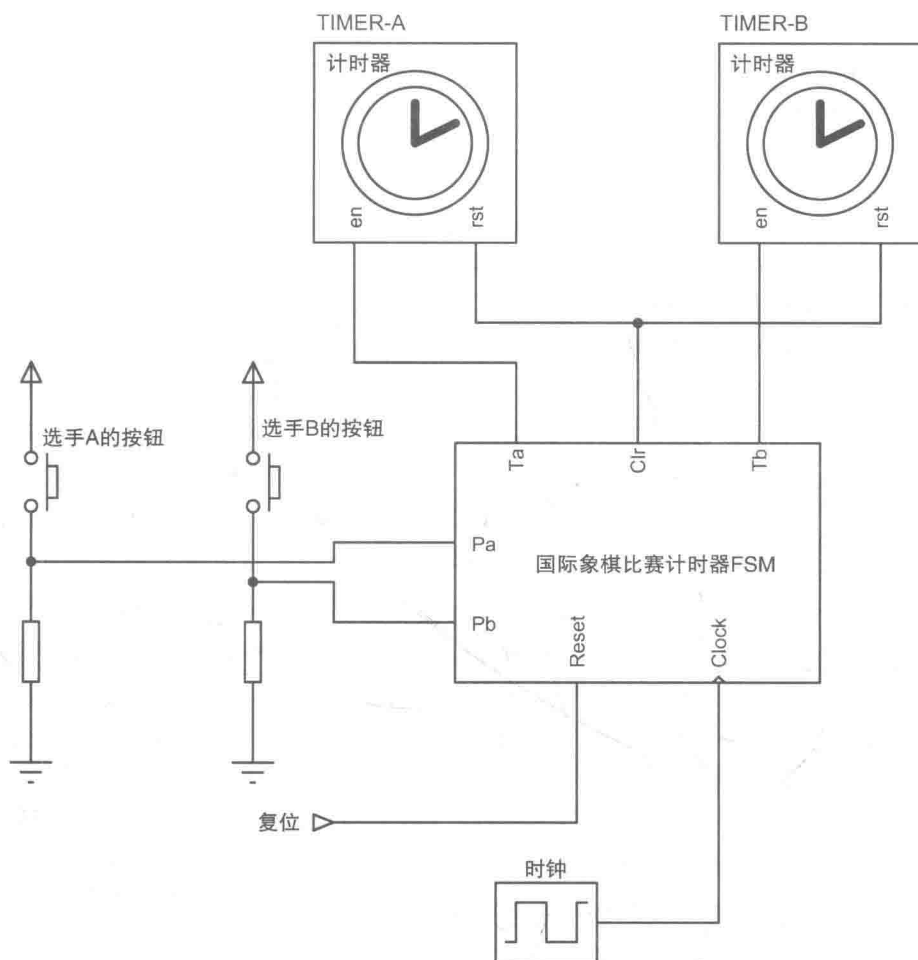


图 8.31 国际象棋比赛计时器系统框图

FSM 控制它们。计时器控制输入信号 en 和 rst 的功能描述如下：

rst——当它为逻辑 1 时，将计时器复位到 0 小时、0 分钟、0 秒。

en——当它为逻辑 1 时，允许计时器从当前所在的时间点开始递增计时。当 en 是逻辑 0，计时停止并保持当前的时间点。

当一场新的比赛开始时，reset 信号会被激活并将系统初始化，两个计时器都将被清零。通过将控制计时器的 FSM 输出信号 clr 拉高，来激活复位两个计时器的 rst 信号。每一个参赛选手都有一个按钮，当按下按钮时，会分别将 FSM 的两个输入 Pa 和 Pb 拉高。时钟复位完成后，比赛开始，后走棋的选手按下按钮，为先走棋的选手开始计时。

例如，如果选手 A 先走棋，此时选手 B 会先按下按钮。此时 FSM 的输出信号 Ta 会被激活，以便通过计时器 A 开始统计选手 A 走第一步棋所花的时间。一旦选

手 A 第一步棋走完，他将按下自己的按钮，计时器 A 此时停止计时，同时选手 B 的计时器开始计时（FSM 输出 Ta 被释放，Tb 被激活）。

为了仿真，假设激活 Pa 和 Pb 所花的时间为至少一个时钟周期，按钮回弹和亚稳态等潜在的问题[3] 暂时在此忽略。

对于两个选手同时按下各自的按钮这种不太常见的现象，FSM 会做出让两个计时器都停止计时的反应。

此时两个计时器均会停在当前的时间，直到按照上述规则恢复比赛，即选手 A（选手 B）按下他的按钮并激活计时器 B（计时器 A）。

图 8.32 是系统状态图。整个系统包含 4 个状态，其名称在圆圈的上半部分显示。输出信号 Ta、Tb 和 Clr 的状态在圆圈的下半部分显示，其中信号名称之前带有“/”符号表示取反，没有则表示信号被拉高。观察各个信号在每个状态里的输出，可以发现象棋比赛计时器 FSM 属于摩尔型 FSM。

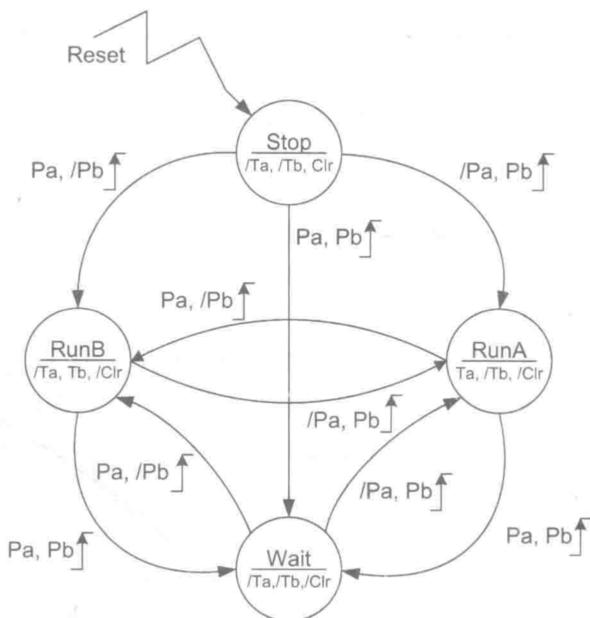


图 8.32　FSM 控制计时器的状态图

输入信号 Pa 和 Pb 的状态，在每一根状态切换箭头旁边都有显示，其格式和 FSM 的输出相类似。FSM 从一个状态转换到下一个状态，是通过时钟信号 Clock 的上升沿触发的。从某个状态转换到别的状态，所对应的输入信号状态组合的个数，如果小于输入状态组合的总数，那么其余所有输入状态组合会导致 FSM 停留在当前这个状态，即下一个状态就是本身。

举例说明，图 8.32 中离开状态 RunA 的条件对应两个输入状态组合 <Pa，Pb> = <1，0>和<1，1>，余下的输入状态组合<0，0>和<0，1>会导致 FSM 停留在 RunA；也就是说，这里存在一个 FSM 停留在 RunA 的可能，这种情况会在输入信号 Pa 为逻辑 0，Pb 为逻辑 0 或者逻辑 1 时出现，这里的 Pb 在 Pa 为逻辑 0 时属于自由态。

异步高有效输入信号 Reset 迫使 FSM 直接进入名为 Stop 的状态，不受其他任何条件影响。

代码 8.7 用 Verilog HDL 描述了图 8.32 里 FSM 的行为

```
1 module chessclkfsm(input reset, Pa, Pb, clock,
```

```
2           output Ta, Tb, Clr);

3 //使用局部参数给状态赋值
4 localparam RunA = 0, RunB = 1, Stop = 2, Wait = 3;

5 reg [1 : 0] state;

6 //状态寄存器和下一状态时序模块组合
7 always @  (posedge clock or posedge reset)
8  begin
9      if (reset)
10      state  < = Stop;
11    else
12      case (state)
13        RunA:
14          casex ({Pa, Pb})
15            2'b0x: state  < = RunA;
16            2'b10: state  < = RunB;
17            2'b11: state  < = Wait;
18          endcase
19        RunB:
20          casex ({Pa, Pb})
21            2'bx0: state  < = RunB;
22            2'b01: state  < = RunA;
23            2'b11: state  < = Wait;
24          endcase
25        Stop:
26          case ({Pa, Pb})
27            2'b00: state < = Stop;
28            2'b01: state  < = RunA;
29            2'b10: state  < = RunB;
              2'b11: state  < = Wait;
31        endcase
32        Wait:
33          if (Pa = = Pb)
34            state  < = Wait;
```

```
35          else if (Pa = =1'b1)
36            state  < =RunB;
37        else
38            state  < =RunA;
39      endcase
40 end

41 //摩尔 FSM 输出赋值
42 assign Ta = state = = RunA;
43 assign Tb = state = = RunB;
44 assign Clr = state = = Stop;
45 endmodule
```

代码 8.7　象棋比赛计时器 FSM Verilog 代码

模块利用局部参数来定义每一个状态。它们在代码第 4 行都被赋予一个整数值。随后模块定义了一个 2bit 位宽的寄存器去存放这些状态，这和代码 8.6 里所用的描述方法是一样的。

从代码第 7 行到第 40 行的 always 时序模块，描述了状态寄存器和下一状态的行为逻辑。由于状态 RunA 和 RunB 之间的转换出现了自由态条件，因此这里用到了 case 语句的一个特殊形式 casex。

使用 casex（第 14 行到第 20 行）来代替 case，使得代码对自由（未知）态（x）的表述更加清晰。事实上，这意味着某一赋值可以对应多个输入条件，例如代码第 14 行和第 15 行可以写成下面的格式：

```
14 case ({Pa, Pb})
15   2'b00, 2'b01 : state  < =RunA;
16 ...
```

运用 case 语句能够将变量 state 每一个可能的情况都考虑到。本例中不需要用 default 来体现其他情况，因为系统状态的数量正好等于 2 的指数。状态 Stop 对应 4 个独立的下一状态，因此需要一个套嵌的 case... endcase 来描述其 4 路分支（代码第 27 行到第 30 行）。case 语句在其 case... endcase 构架内部对应的每一条语句的优先级都是相同的，因此内部每一条语句之间必须没有重叠或者相互独立。在这之前，单个分支是可以被多次赋值的，只要这些数值不在其他分支的赋值语句内出现就不会报错。

Wait 状态是用嵌套的 if... else 语句来描述的，这里换一种方式的目的是向大家展示 Verilog 语句的灵活性。代码第 33 行到第 38 行和图 8.32 所反映的状态切换是相吻合的，但是读者必须注意输入条件中含有状态停留条件 <Pa, Pb> = <0, 0> 和 <1, 1>。

而且大家还需要考虑到，尽管 if... else... if 语句有一定的优先级，但是经过综合工具后生成逻辑电路的过程，是不会将这些优先级纳入考虑的。这是因为 if... else 构架中每一条语句所对应的条件是相互独立的。

FSM 的输出信号 Ta、Tb 和 Clr 是摩尔型，即只和 FSM 的当前状态有关。代码第 42 行到第 44 行用连续赋值语句来描述它们。每一个输出的赋值条件，都是将状态寄存器变量 state 和局部参数的值不断地进行比较，一旦输出在对应的状态被激活，便立刻赋值。

本例中，每一个输出只对应一个状态，因此，这意味着输出端的逻辑电路，只比一个与门稍微复杂一点而已。

这里的输出端逻辑电路，还可以进一步被简化成将输出信号和其状态进行配对编码。对于每一个状态来说，相应的输出都是独立的，因此只要将状态值定义成和输出一样便可，即用代码 8.8 来代替之前代码 8.7 里面的局部变量。

```
3 //状态赋值和输出 Ta、Tb 和 Clr 相一致
4 localparam RunA =3'b100,
5         RunB =3'b010,
6         Stop =3'b001,
7         Wait =3'b000;

8 reg [2 : 0] state; //状态的位宽和输出的个数一样
  ...
39    default : state  < =3'bx;
40 endcase
  ...
41 //输出等于状态位数
42 assign Ta =state [2];
43 assign Tb =state [1];
44 assign Clr =state [0];
```

代码 8.8　另一种状态赋值和输出相匹配的方法

代码 8.8 第 42 行到第 44 行，可以取代代码 8.7 中相应位置的输出端连续赋语句。现在很明显可以看出，每一个输出信号都直接映射状态寄存器相应的一位。

上述改变状态赋值的方法，有一个需要注意的地方，就是需要将状态寄存器位宽和输出信号的个数相匹配。代码 8.8 的第 8 行声明状态寄存器 state 的位宽是比特（3bit）。因此，下一状态行为描述的代码在第 39 行必须加入一个默认语 default，这样，多出来的没有用到的状态（$2^3-4=4$）才算是被考虑到了。

图 8.33 用一个比较简单的测试模块，来对象棋比赛计时器 FSM 进行了仿真

图中也给出了仿真结果波形图。其中输入、输出和状态变量之间的关系，也符合之前的设计初衷。大部分 Verilog 仿真工具可以提供一个可视化功能，将每一个状态的名称在波形图中显示出来，而不是显示每个状态对应的赋值，这样更为直观。这给分析、理解和验证 FSM 的行为，提供了很重要的视觉辅助功能。

```
1  `timescale 1 ms / 1 ms
2  module Test_chessclkfsm();

3  reg RES, A, B, CLK;
4  wire Ta, Tb, Clrt;

5  //产生10Hz的时钟
6  initial
7  begin
8    CLK = 1'b0;
9    forever
10     #50 CLK = ~CLK;
11 end

12 //产生输入信号
13 initial
14 begin
15   RES = 1'b1; A = 1'b0; B 1'b0;
16   #200 RES = 1'b0;
17   #200;
18   A = 1'b1;
19   #550 A = 1'b0;
20   #350 B = 1'b1;
21   #750 B = 1'b0;
22   #400;
23   A = 1'b1; B = b1;
24   #350;
25   A = 1'b0; B = 1'b0;
26   #450;
27   A = 1'b1;
28   #800;
29   $stop;
30 end

31 // FSM建模
32 chessclkfsm mut (.reset (RES),
33                  .Pa (A), . Pb(B), . clock (CLK),
34                  .Ta (Ta), .Tb (Tb), .Clr (Clrt));
35 endmodule
```

图 8.33　象棋比赛计时器 FSM 仿真代码和波形图

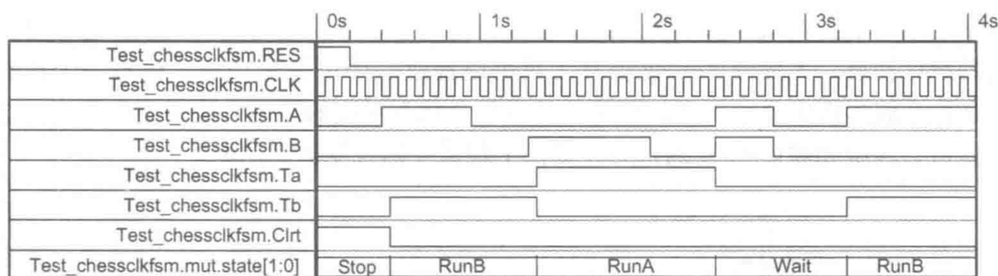


图 8.33 象棋比赛计时器 FSM 仿真代码和波形图（续）

8.7.2 实例 2：带有自动落锁功能的密码锁 FSM

第二个运用 FSM 设计的系统看起来更加复杂，其中包含了好几个模块，组合逻辑和时序逻辑都有涉及。本例还向大家展示了 FSM 是如何跟其他同步时序模块进行互动，这些同步时序模块都用同一个时钟驱动，且描述方式为行为模式。

图 8.34 是“数字密码锁”的系统框图。系统的中间部分是一个 FSM，FSM 模块用控制器来标注，它的功能是检测用户是否通过左边键盘开关上的按钮，输入正确的 4 位密码。

系统向用户提供 8 个低有效按钮（SW［0］…SW［7］)。前 4 个（SW［0…3］）通过一个 4 选 1 多路选择器连接到系统，它们代表的是代码切换。由于这 4 个按钮是否通过多路选择器连接到系统，是用户根据解锁密码决定的。这样，密码就通过硬件方式和系统形成了固定连接。

和按钮相连的 8 输入与门产生一个输出信号名为 allsw，如果任何一个按钮被按下，此信号将被输出逻辑 0。4 选 1 多路选择器的输出 mux _ out，会根据其地址选择输入信号 sel［0..1］所对应的按钮被按下后输出逻辑 0。这样，多路选择器可以按照一定的次序来选择每一个按钮，而且输出信号 mux _ out 只有在正确的按钮被按下时才会被拉低。

所有的按钮都是异步输入信号，而密码锁系统是同步运行的。对于系统来说它无法估计任何一个按钮会在什么时刻被按下，因此，信号 mux _ out 和 allsw 所生的逻辑 0 脉冲宽度也是无法估计的。如果上述信号是直接进入 FSM 的，那么一个按钮在按下后释放所需的时间，将被系统识别为 n 个输入信号，如果释放某按钮所需的时间是 0.5s，那么 n 的表达式为：n =0.5/系统时钟周期。

上述按钮回弹问题可以用图 8.35 所示的“边沿检测”电路来解决。系统运了两个这种电路模块，分别命名为 DET1 和 DET2。通过分析图 8.35，可以看出种电路，其实就是一个同步 2 比特（2bit）移位寄存器，其中第一个触发器的输和第二个触发器的输出取反后接入一个与门，产生整个电路的输出信号。

这种类似简易电路的模块可以提供同步和检测沿的双重功能，当输入信edge _ in 从逻辑 1 变为逻辑 0 的那一刻，模块会输出一个和单个时钟周期同样宽

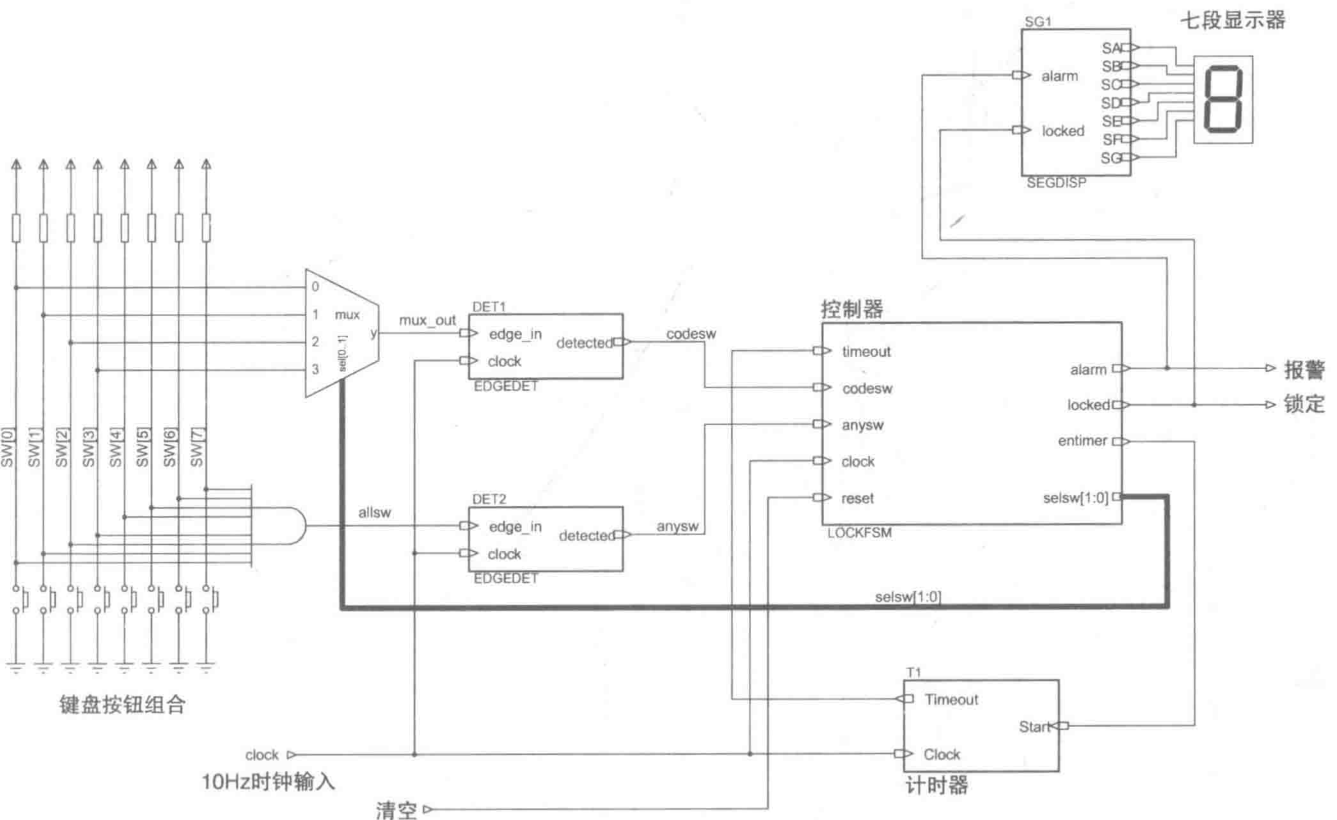


图 8.34　密码锁系统框图

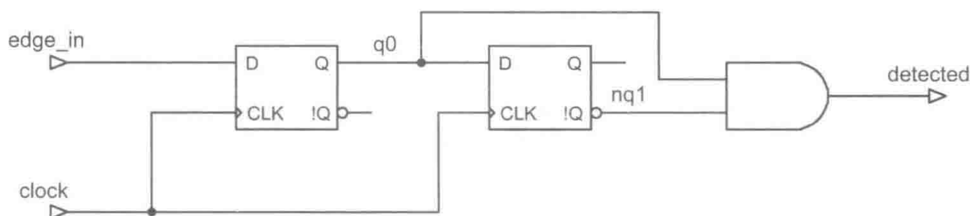


图 8.35 边沿检测电路 edgedet 模块

的逻辑 1 脉冲信号，并命名为 detected（查明），不管之后输入信号 edge _ in 会将逻辑 0 保持多长时间。

当需要建立异步电路和同步电路之间的通信时，在不考虑系统亚稳态[3]的情况下，图 8.35 都会有效解决 FSM 能够及时检测到按钮被按下的问题。

两个边沿检测模块的输出 codesw 和 anysw 被直接导入状态机模块 LOCKFSM。事实上两个边沿检测模块和 FSM 都是由同一个时钟驱动的，这就使得两种不同类型的模块形成了同步。当某一个按键被按下时，并且是正确的（即 4 选 1 复位选择器正好选择的是这个按键），那么 FSM 模块 LOCKFSM 会在同一个时钟周期内，收到 codesw 和 anysw 两个正脉冲。收到这两个脉冲会告诉 FSM 此时按下的是正确的密码，并触发 FSM 进入下一个状态。

代码 8.9 和代码 8.10 分别给出了的 D 触发器和边沿检测模块的 Verilog 描述。

```
1 module dff (output reg q, input d, clk);
2 always @  (posedge clk) q< =d;
3 endmodule
```

代码 8.9 D 触发器的 Verilog 描述

```
1 module edgedet (input edge _ in,
2             output detected,
3            input clock);
4 wire q0, q1;
5 dff dff0 (.q (q0), .d (edge _ in), .clk (clock));
6 dff dff1 (.q (q1), .d (q0), .clk (clock));
7 assign detected=q0 & ~q1;
8 endmodule
```

代码 8.10 边沿检测模块的 Verilog 描述

图 8.34 中还有一个计时模块（TIMER），标记为 T1。这个模块和 FSM 模块过信号 entimer（计时使能）和 timeout（计时超时）两个信号相连，并且和 FSM 块、边沿检测模块一样，通过同一个系统时钟驱动，确保了整个系统是同步的。

计时器的功能是提供一个自动锁定的机制，在系统进入解锁状态 30s 后自动系统回到锁定状态。

系统时钟的频率是 10Hz，因此这里的计时器需要计数到十进制 300，代

8.11 描述了计时器的行为模式。

```
1 module Timer (input Clock, Start, output Timeout);
2 //延迟时间换算成系统时钟的脉冲数
3 localparam NUMCLKS =300;

4 reg [8 : 0] q;
5 always @  (posedge Clock)
6 begin
7   if (! Start  | |  (q = =NUMCLKS))
8     q  < =9' b0;
9   else
10    q  < =q+1;
11 end
12 //解码计数器的输出
13 assign Timeout =  (q = =NUMCLKS);

14 endmodule
```

代码 8.11　自动锁定计时器 Verilog 描述

计时器模块 TIMER 的行为完全是同步的：当输入信号 Start 处于逻辑 0 状态时，计时器不工作，且寄存器 q 的值始终为 0。

系统 FSM 在进入解锁状态后激活信号 entimer（此信号和计时器模块的 Start 相连），随后寄存器 q 开始在每一个时钟上升沿到来时计数，直到和数值 NUMCLKS（300_{10}）相等，此时信号 Timeout 输出逻辑 1，脉冲宽度为一个时钟周期，随后计数器清零。

FSM 在接收到信号 timeout 变为逻辑 1 之后，会返回到状态 s0，并且 FSM 输出信号 locked 被拉高。回到状态 s0 后，FSM 同时将输出信号 entimer 拉低，让计时器停止工作，等待下一次需要计数的时刻到来。

到这里为止，系统里就剩下七段显示解码模块 SEGDISP 还没有介绍了。此模块由组合逻辑电路构成，驱动一个低有效的七段显示器单元，用于显示系统的运行状态。显示值取决于 FSM 的输出信号 alarm 和 locked："L" 表示锁定，"U" 表示解锁，"A" 表示报警。代码 8.12 描述了显示器的行为模式。

```
1 module segdisp (input locked, alarm,
2          output SA, SB, SC, SD, SE, SF, SG);
3 reg [6: 0] seg;

4 always @  (locked or alarm)
5 begin
6   if (alarm = =0)
```

```
7      seg =7'b0001000; //显示“A”
8    else if (locked= =0)
9      seg =7'b1000001; //显示“U”
10   else
11     seg =7'b1110001; //显示“L”
12 end

13 assign {SA, SB, SC, SD, SE, SF, SG} =seg;

14 endmodule
```

代码 8.12 七段显示解码模块 Verilog 描述

图 8.36 是密码锁系统中央 FSM 模块 lockfsm 所对应的状态图。

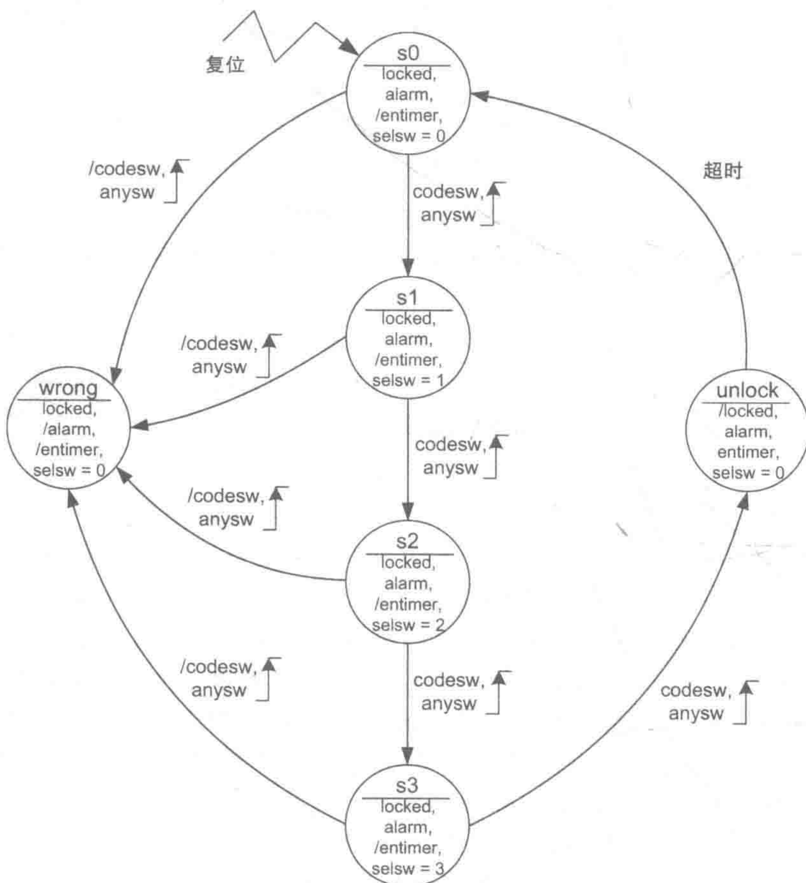


图 8.36 密码锁 FSM（lockfsm）状态图

FSM 通过异步复位输入信号进行初始化，将 FSM 复位到状态 s0，此时输出信号 locked 和 alarm 的值都为逻辑 1，表示系统被锁定并且不在报警状态（alarm 信号是低有效）。2 位输出信号 selsw 被设为逻辑 0，因而按照顺序从 4 选 1 复位选择

中指向第一个按钮。由于 entimer 信号被设为逻辑 0，计时器此时也是处于复位状态。

系统下一步的动作取决于按下哪一个按钮。如果第一个按钮（SW [0]）被按下，那么 FSM 输入信号 codesw 和 anysw 将同时拉高，FSM 此时会进入状态 s1，并在此等待下一个按钮被按下。

在状态 s1，状态机输出 selsw 被设为 1，因而此时将复位选择器和第二个按钮相连，这样按照次序，此时按钮 SW [1] 处于代码序列中。如果这时候用户按下 SW [1]，会将信号 codesw 和 anysw 分别再次拉高，将 FSM 代入状态 s2。

进入状态 s2 之后，selsw 的值被 FSM 设为 2，此时选择的是复位选择器的第三个输入，它和按钮 SW [2] 相连。

根据上述描述，用同样的方法，按下 SW [2] 按钮后再按下 SW [3]，将会使 FSM 进入解锁（unlock）状态。这里进入解锁状态的条件，是用户依次按下 4 个按钮（SW [0] …SW [3]）的顺序是正确的。进入解锁状态后，输出信号 locked 变为逻辑 0，且七段显示器会显示字母“U”。

根据图 8. 36，此时 FSM 的输出信号 entimer 被激活，计数器开始工作。FSM 在输入信号 timeout 维持在逻辑 0 的时间范围内，都会停留在解锁状态（并且此时整个系统的异步复位信号已经被释放）。

之前已经讨论过，FSM 处于解锁状态的时间是 300 个时钟周期或者 30s。时间一到，FSM 会立刻回到 s0，并将信号 locked 拉高。

从状态 s0 到 s3 转换的过程中，如果任何时候按下一个错误的按钮，那么通过 8 输入与门，从信号 anysw 端会产生一个脉冲，但是从信号 codesw 端却收不到脉冲，因为当前复位选择器所对应的输入端按钮没有被按下（拉低）。

图中在某些情况下，FSM 会进入错误（wrong）状态，表示用户没有按下指定的按钮。FSM 进入此状态后，低有效输出信号 alarm 将被激活，七段显示器会显示字母“A”。

进入错误（wrong）状态之后是无法离开的，因为没有任何输入条件可以触发 FSM 离开此状态，即只有通过强制复位，整个系统才能使得 FSM 离开错误状态。因此，很明显，系统的输入信号 clear 必须带有安全防护措施，只有授权的操作员才能复位系统。

代码 8. 13 是 FSM 的行为描述

```
1 module lockfsm (input clock, reset,
2         codesw, anysw,
3         output reg [1: 0] selsw,
4         output locked, alarm, entimer,
5         input timeout);
```

```
6 localparam s0 =3'b000, s1 =3'b001, s2 =3'b010,
7         s3 =3'b011,
8         wrong =3'b100, unlock =3'b101;

9 reg [2: 0] lockstate;

10 always @  (posedge clock or posedge reset)
11 begin
12 if (reset = =1'b1)
13  lockstate  < =s0;
14 else
15  case (lockstate)
16     s0: if (anysw & codesw)
17      lockstate  < =s1;
18     else if (anysw)
19      lockstate  < =wrong;
20     else
21      lockstate  < =s0;
22     s1: if (anysw & codesw)
23      lockstate  < =s2;
24     else if (anysw)
25      lockstate  < =wrong;
26     else
27      lockstate  < =s1;
28     s2: if (anysw & codesw)
29      lockstate  < =s3;
30     else if (anysw)
31      lockstate  < =wrong;
32     else
33      lockstate  < =s2;
34     s3: if (anysw & codesw)
35      lockstate  < =unlock;
36     else if (anysw)
37      lockstate  < =wrong;
38     else
39      lockstate  < =s3;
40     wrong: lockstate  < =wrong;
41     unlock: if (timeout)
42      lockstate  < =s0;
```

```
43     else
44      lockstate  <=unlock;
45     default : lockstate  <=3'bx;
46   endcase
47 end

48 always @  (lockstate)
49 begin
50   case (lockstate)
51     s0: selsw=0;
52     s1: selsw=1;
53     s2: selsw=2;
54     s3: selsw=3;
55     wrong: selsw=0;
56     unlock: selsw=0;
57     default : selsw=2'bx;
58   endcase
59 end

60 assign locked =  (lockstate= =unlock)? 0: 1;
61 assign alarm =  (lockstate= =wrong)? 0: 1;

62 assign entimer =  (lockstate= =unlock)? 1: 0;

63 endmodule
```

代码 8.13　密码锁 FSM Verilog 描述

与之前的例子一样，这是一个摩尔 FSM。因此从代码第 10 行开始的 always 时序模块，只描述状态寄存器和下一状态的行为。

输出信号赋值是用 always 组合逻辑（代码第 48 行到第 59 行）和连续赋值语句（代码第 60 行到第 62 行）来完成的。代码第 9 行定义了 3 位状态寄存器 lockstate，代码第 6 行用局部参数将所有 6 个状态全部依次编号。

代码第 45 行和第 57 行将没有用到的状态，用 default 语句进行声明，定义为自由态，确保了代码的严密性。

除了状态 wrong，其他所有状态都运用 if…else 语句来描述状态转换逻辑。例如，对于状态 s1，我们将代码重复如下：

```
s1 : if (anysw & codesw)
       lockstate  <=s2;
     else if (anysw)
```

```
        lockstate <=wrong;
    else
        lockstate <=s1;
```

这里首先测试条件 anysw & codesw 是否成立；只有当两个信号均为逻辑 1 时，条件才能满足。如果条件成立，FSM 会从状态 s1 进入状态 s2。如果第一个条件无法满足，那么可能的情况是其中一个是逻辑 0，或者两个都为逻辑 0。这段代码暗示着信号 codesw 在 anysw 为逻辑 0 时，是不可能为逻辑 1 的，所以只需要判断此时的 anysw 是否为 1。如果条件成立，代表用户按下的按钮不对，FSM 随后会报警。

如果没有任何按钮被按下，FSM 会停留在当前状态，即状态 s1。上述语句最后一个 else 分句实现了这一功能。

图 8.34 所对应的整个密码锁系统的 Verilog 描述代码如下：

```
1 module comblock (input clock, clear,
2         input [7:0] switches,
3         output alarm, locked,
4         output SA, SB, SC, SD, SE, SF, SG);

5 wire mux_out, anysw, codesw,
6   allsw, entimer, timeout;

7 wire [1:0] selsw;

8 //4 选 1 多路选择器
9 assign mux_out =selsw ==0? switches [0]:
10            (selsw ==1? switches [1]:
11            (selsw ==2? switches [2]:
12            (selsw ==3? switches [3]: 1'b0)));

13 //所有按钮对应的与门
14 assign allsw =&switches;

15 edgedet det1 (.edge_in (mux_out),
16          .detected (codesw),
17          .clock (clock));

18 edgedet det2 (.edge_in (allsw),
19          .detected (anysw),
20          .clock (clock));

21 Timer t1 (.Clock (clock),
```

```
22          .Start (entimer),
23          .Timeout (timeout));

24 lockfsm controller (.clock (clock),
25          .reset (clear),
26          .codesw (codesw),
27          .anysw (anysw),
28          .selsw (selsw),
29          .locked (locked),
30          .alarm (alarm),
31          .entimer (entimer),
32          .timeout (timeout));

33segdisp sg1 (.locked (locked),
34          .alarm (alarm),
35          .SA (SA),
36          .SB (SB),
37          .SC (SC),
38          .SD (SD),
39          .SE (SE),
40          .SF (SF),
41          .SG (SG));
42 endmodule
```

代码 8.14　完整的密码锁系统 Verilog 描述

模块 comblock 的描述涵盖了前面所有介绍的模块，同时还用两条连续赋值语句描述了 4 选 1 多路选择器和 8 输入与门（代码第 9 行和第 14 行）。

密码锁的系统仿真流程，用名为 test _ comblock 的测试模块代码未完成，如代码 8.15 所示：

```
1 `timescale 1ms/1ms
2 module test _ comblock ();

3 //输入
4 reg clock;
5 reg clear;
6 reg [7: 0] switches;

7 //输出
8 wire alarm;
9 wire locked;
```

```
10 wire SA, SB, SC, SD, SE, SF, SG;

11 //密码锁建模
12 comblock UUT (
13             .clock (clock),
14             .clear (clear),
15             .switches (switches),
16             .alarm (alarm),
17             .locked (locked),
18             .SA (SA), .SB (SB), .SC (SC),
19             .SD (SD), .SE (SE), .SF (SF), .SG (SG)
20             );

21 initial
22 begin
23   clock =1'b0;
24   forever
25     #50 clock = ~clock;
26 end

27 initial
28 begin
29   clear =1'b1;
30   switches =8'b11111111;

31   repeat (3) @  (negedge clock);
32   clear =1'b0;

33   repeat (3) @  (negedge clock);
34   switches [0] =1'b0;

35   repeat (2) @  (negedge clock);
36   switches [0] =1'b1;

37   repeat (3) @  (negedge clock);
38   switches [1] =1'b0;

39   repeat (2) @  (negedge clock);
40   switches [1] =1'b1;
```

```
41  repeat (3) @  (negedge clock);
42  switches [2]  =1'b0;

43  repeat (2) @  (negedge clock);
44  switches [2]  =1'b1;

45  repeat (3) @  (negedge clock);
46  switches [3]  =1'b0;

47  repeat (2) @  (negedge clock);
48  switches [3]  =1'b1;

49  repeat (400) @  (negedge clock);    //等待计时结束

50  clear =1'b1;

51  repeat (4) @  (negedge clock);
52  clear =1'b0;
53  repeat (3) @  (negedge clock);
54  switches [0]  =1'b0;

55  repeat (2) @  (negedge clock);
56  switches [0]  =1'b1;

57  repeat (3) @  (negedge clock);
58  switches [5]  =1'b0;

59  repeat (2) @  (negedge clock);
60  switches [5]  =1'b1;

61  repeat (3) @  (negedge clock);
62  switches [2]  =1'b0

63  repeat (2) @  (negedge clock);
64  switches [2]  =1'b1;

65 repeat (3) @  (negedge clock);
66  switches [3]  =1'b0;

67  repeat (2) @  (negedge clock);
68  switches [3]  =1'b1;
```

```
69   repeat (4) @  (negedge clock);
70   clear =1'b1;

71   repeat (4) @  (negedge clock);

72    $stop;
73 end

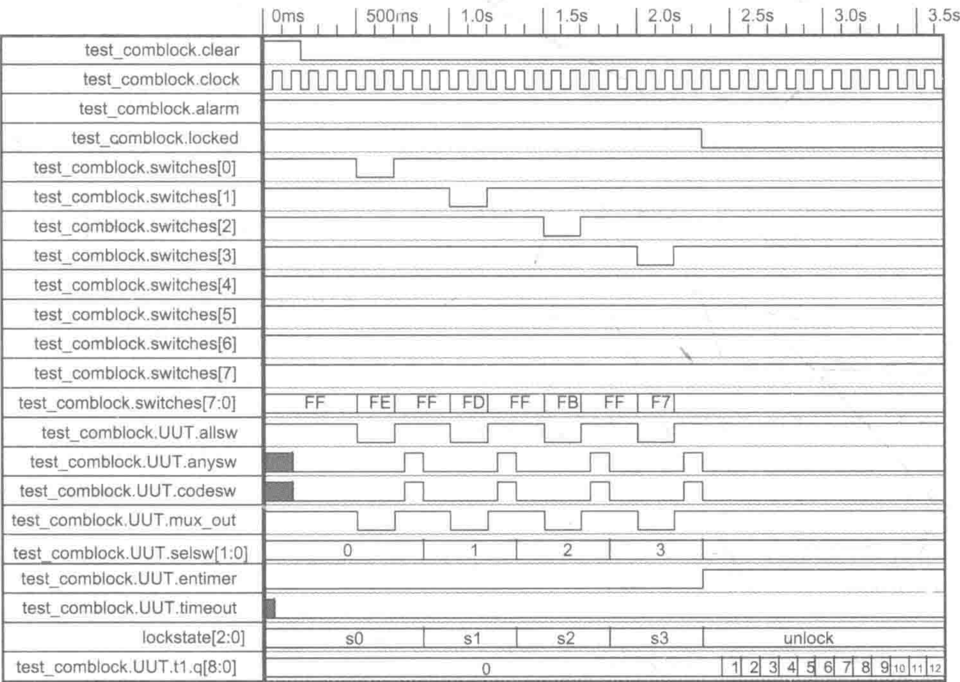
74 endmodule
```

代码 8.15　密码锁系统仿真测试代码

代码第 21 行用 initial 时序语句产生一个 10Hz 的时钟。

代码第 27 行开始的第二个 initial 模块语句，模拟了正确的按钮按下的顺序，目的在于测试系统能否正常进入 unlock 状态。随后仿真程序延迟 40s，这里延迟用的是 repeat 循环语句，以便观察自动上锁功能。最终，在复位系统之后，程序模拟了一个错误的按钮输入顺序，用来验证系统能否进入报警状态。

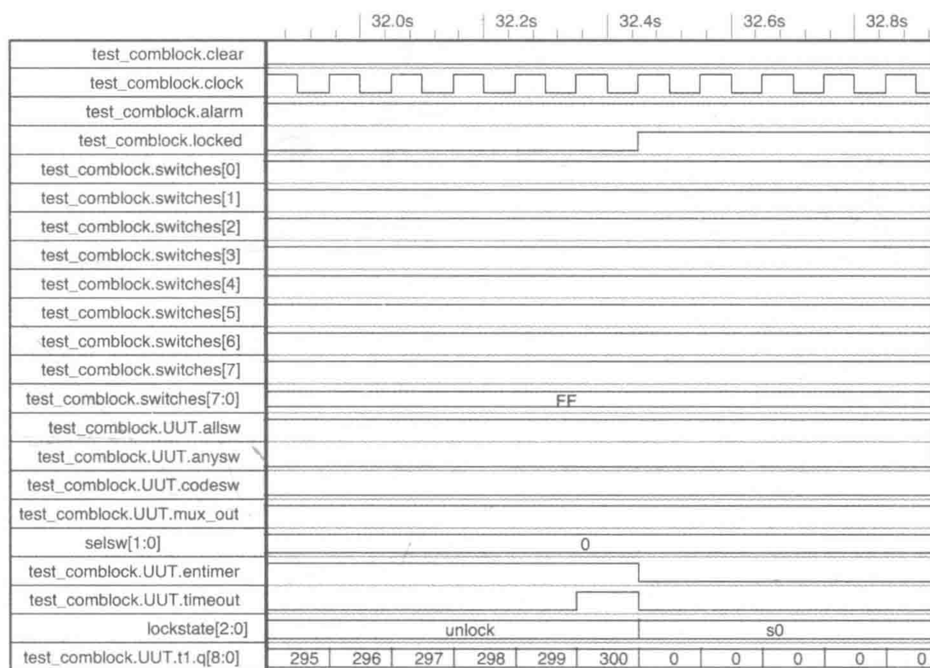
图 8.37 对应的是不同情况下的仿真波形图。



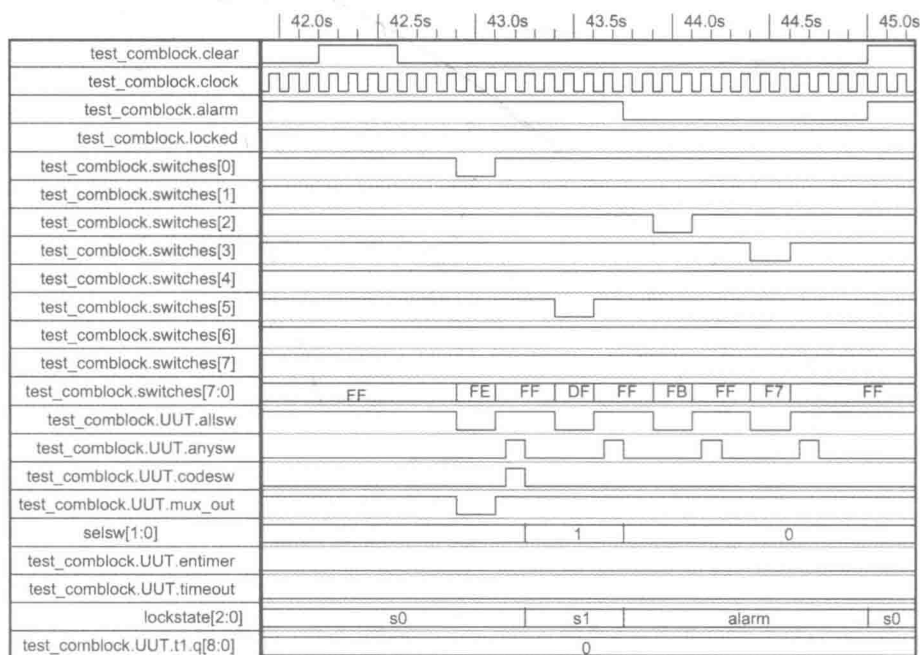
a)

图 8.37　密码锁仿真波形图

a）正确按钮顺序



b)



c)

图 8.37　密码锁仿真波形图（续）

b）自动上锁功能　c）错误按钮输入顺序

参考文献

1. Ciletti MD. Modeling, Synthesis and Rapid Prototyping with the Verilog HDL. New Jersey: Prentice Hall, 1999.
2. Ciletti MD. Advanced Digital Design with the Verilog HDL. New Jersey: Pearson Education, 2003 (Appendix I – Verilog-2001).
3. Wakerly JF. Digital Design: Principles and Practices, 4th edition. New Jersey: Pearson Education, 2006 (Metastability and Synchronization, Section 8.9).

第 9 章　异步 FSM

9.1　概述

大部分 FSM 系统是同步的，意味着它们依靠时钟驱动系统，从一个状态转换到下一个（或另一个）状态。使用时钟来控制状态之间的同步切换，使得 FSM 的硬件电路在下一个转换到来之前，有足够的时间趋于稳定，因此避免了逻辑门之间的延迟现象对系统造成影响。正因为如此，同步系统到目前为止是最常见的数字电路系统，且大部分硬件描述语言也根据同步系统的特性，进行了专门的优化。

然而，还有另一种 FSM，它不用时钟触发状态转换，这就是所谓的异步 FSM。在异步 FSM 中，状态的切换是由事件触发的，FSM 不需要等待固定时钟脉冲的到来。因此，异步 FSM 有时候被称为“事件触发”FSM。

图 9.1 是一个比较典型的异步 FSM。其中，当输入信号 s 变为逻辑 1，并且输入信号 c 变为逻辑 0 时，FSM 会从状态 s0 进入到状态 s1。到达状态 s1 之后，FSM 会在此停留，直到输入信号 c 变为逻辑 1 时，才会进入状态 s2。此时，当输入信号 c 再变为逻辑 0 时，FSM 会进入状态 s3，回到状态 s0 的条件是输入信号 s 变为逻辑 0。

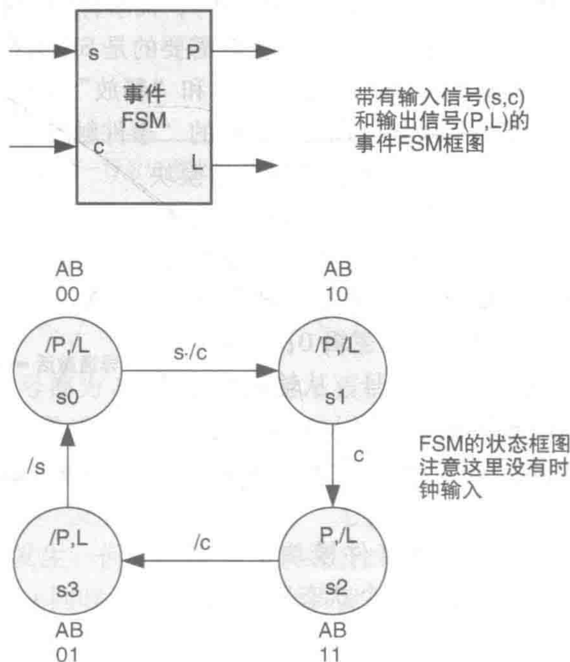


图 9.1　异步 FSM 举例

这里，FSM 只有在输入信号发生变化时才切换状态，即事件触发的本质。

某些情况下，即使没有输入信号变化也需要切换状态（此时就是时钟驱动系统的原理了）。

图 9.2 中，从状态 s3 到状态 s0 是没有输入信号驱动的。这意味着当 FSM 到达状态 s3 时（输入信号 x 变为逻辑 1），它将立刻回到状态 s0。而从状态 s3 回到状态 s0 所花的时间，是由系统传输延迟决定的。此时转换速度和逻辑

器件状态翻转的速度一样。

事件触发的 FSM 有一个非常重要的特性，就是当它处于稳定状态时（可能此时在等待下一个触发事件），CMOS 电路的功耗是非常小的，因为此时系统里没有时钟在运行，几乎不产生能耗。这种特性让异步系统保证运行速度的同时，也只产生很低的功耗。至于系统状态切换的速度能够有多快，也是由参与产生触发事件的逻辑电路传输延迟决定的，一旦满足条件，状态会立刻切换。

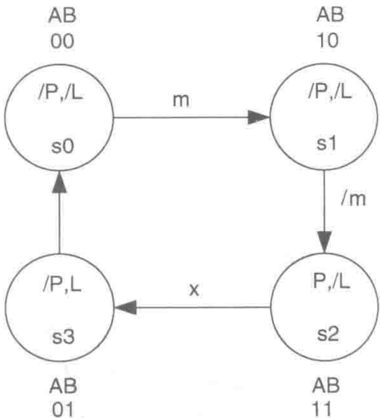


图 9.2 没有事件触发的状态切换

9.2 事件触发逻辑的设计

根据上一节的阐述，事件触发的系统框图和时钟触发的 FSM 系统框图有很多类似的地方。然而，当有时钟驱动时，系统综合会使用不同种类的触发器（D 触发器，T 触发器或者 JK 触发器），而事件触发系统需要利用存储单元，它们不需要时钟驱动。这意味着或许此时需要的是 SR 锁存器。但是在实际应用中，某些场合锁存器需要多个“激活”（s）和“释放”（r）输入信号。随后可以针对不同的情况，推导一系列公式实现常规的“事件触发”单元。

图 9.3 是一个事件触发的模块框图。图中有一个“状态激活”输入信号，可以将输出变为逻辑 1；另一个是“状态释放”输入信号，可以将输出变为逻辑 0；还有一个“状态保持”信号，从输出信号端引出来作为反馈输入，目的是让模块保持当前的“激活”或者“释放”状态。

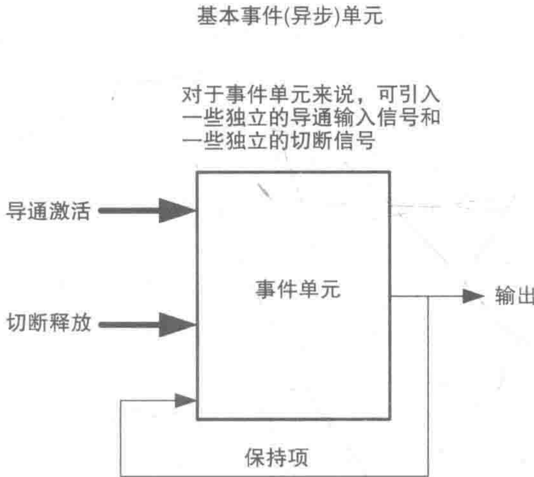


图 9.3 事件模块框图

为了推导事件模块的逻辑公式，首先需要一个状态真值表，如表 9.1 所示。其中，“状态激活”输入信号在表格里用 s 来表示，“状态释放”用 r 来表示，输出的当前状态为 Q_n，输出的下一个状态为 Q_{n+1}。个输入 s 和 r，以及输出的当前状态 Q_n 在表格里用二进制表示。因此模块所有可出现的状态均比较直观，但是 Q_{n+1} 的状态需要自己判定。

第 1 行，s = r = 0，模块处于复位状态。由于模块是能够维持当前状态的，

此当 s = r = 0 时，$Q_{n+1} = Q_n = 0$。

第 2 行，s = r = 0，但是此时模块属于激活状态。因此，$Q_{n+1} = 1$，而且模块会在之后维持激活状态。

第 3 行，s = 0，但是 r = 1，意味着模块触发一个复位的动作。由于模块本身已经处于复位状态，所以 $Q_{n+1} = 0$。

第 4 行，s = 0，r = 1，和之前一样，但是模块此时是激活的状态，因此下一个状态 Q_{n+1}的结果是逻辑 0。

表 9.1　事件模块的状态表

行数	s	r	Q_n	Q_{n+1}事件
1	0	0	0	没有变化
2	0	0	1	没有变化
3	0	1	0	强制释放
4	0	1	1	状态释放
5	1	0	0	状态激活
6	1	0	1	强制激活
7	1	1	0	这种情况不成立
8	1	1	1	这种情况不成立

注：Q_n是当前状态，Q_{n+1}是 Q 的下一个状态。每一行对应模块所处的状态，并可以用来决定事件模块的“特征方程”。

第 5 行，s = 1，r = 0，模块当前是释放的，所以下一个状态 $Q_{n+1} = 1$。

第 6 行，s = 1，r = 0，和之前一样，模块是激活的，因此下一个状态会继续保持，即 $Q_{n+1} = 1$。

第 7 行和第 8 行，s 和 r 两个信号都为 1。这种情况对于模块来说有点不切实际，此时两个输入信号是冲突的（即激活模块同时又释放模块，同时进行!）。很明显，这不可能。所以根据常理，这两行下一状态被定义为“不成立”。这里意味着第 7 和第 8 行所对应的情形“不会”发生，通常称之为“无关”状态。不过需要提醒的是这类“无关”状态不会发生，但在推导异步系统公式时，这样的假设需要被纳入考虑范围。输入状态 s 和 r 同时为 1 的情况不允许发生。在输出端的反映就是 Q_{n+1}的状态，在第 7 和第 8 行就是 x。

表格 9.2 为完整的状态表，Q_{n+1}的公式可以根据表格中 s、r 和 Q_n 的状态来推导

$$
\begin{aligned}
Q_{n+1} &= /s \cdot /r \cdot Q_n + s \cdot /r \cdot /Q_n + s \cdot /r \cdot Q_n + s \cdot r \cdot Q_n + s \cdot r \cdot /Q_n \\
&= /s \cdot /r \cdot Q_n + s \cdot /r + s \cdot r \\
&= /s \cdot /r \cdot Q_n + s
\end{aligned}
$$

表 9.2 时间模块的完整真值表

s	r	Q_n	Q_{n+1}	
0	0	0	0	没有变化
0	0	1	1	没有变化
0	1	0	0	强制释放
0	1	1	0	释放（复位）
1	0	0	1	激活
1	0	1	1	强制激活
1	1	0	x	无关
1	1	1	x	无关

运用辅助定律将公式进一步整理，得到下面的时序公式：$Q_{n+1} = s + Q_n/r$

推导出来的时序公式看作事件模块的“特性方程”。注意到原本的公式里 s. /r + s. r被简化为 s。

时序公式可以用下面文字来表达：新的模块输出状态等于当前状态 Q_n和释放输入信号 r 取反后相与的结果，再和激活输入信号 s 相或而得出。

要证明公式也很简单，只要给 s、r 和 Q_n设定初始值，然后通过公式计算 Q_{n+1} 的结果和真值表来比较验证。注意到，公式里 r 这一项需要先取反变成/r，所以 r = 0意思是/r = 1，r = 1 意思是/r = 0。

计算过程如下：当 s = 1，r = 0，Q_n = 0，则 Q_{n+1} = 1 + 0. 1 = 1，即模块被激活（输出从 0 变为 1）；

当 s = 0，r = 0，Q_n = 1，则 Q_{n+1} = 0 + 1. 1 = 1，模块保持激活状态（输出保持逻辑 1）；

当 s = 0，r = 1，Q_n = 1，则 Q_{n+1} = 0 + 1. 0 = 0，模块被复位（下一个状态输出从 1 变为 0）；

当 s = 0，r = 0，Q_n = 0，则 Q_{n+1} = 0 + 0. 1 = 0，模块保持释放（下一个状态输出继续保持 0）。

如上所述，时序公式在这里稍微有点受限，因为模块里只有两个输入 s 和 r 实际情况是，通常的事件触发系统需要多个激活信号和释放信号，这样系统可以满足不同的激活和释放条件。但是这些统统需要用逻辑或运算来实现，因为系统状态是连续的，同一时刻只能处理一种激活和释放的组合条件。所以时序公式可以通过引入多个输入信号来进行优化：

多路激活信号 $\sum s_r = s_1 + s_2 + \cdots + s_n$，其中 s_n是激活输入信号的最终项。

多路释放信号 $\sum s_r = r_1 + r_2 + \cdots + r_n$，其中 r_n是释放输入信号的最终项。

所以时序公式变为

$$Q_{n+1} = \sum s_Q + Q_n \cdot \sum /r_Q \quad (9.1$$

这就是事件模块所对应时序公式的最终形式。也被称之为与非时序公式[1]。

与之对应的还有一个被称之为或非时序公式的可以定义为

$$Q_{n+1} = (\sum s_Q + Q_n) \cdot \sum /r_Q \tag{9.2}$$

但这个公式现在已经不是很常用了。读者有兴趣可以自己推导或非时序公式（提示：将$\sum s_Q$和（$\sum r_Q + \sum /r_Q$）相与，然后展开）。

如果将事件命名为 A，式（9.1）可以写成

A = ∑（所有激活 A 的输入信号）·A + ∑/（所有释放 A 的输入信号）

式（9.2）可以写成

A =（∑（所有激活 A 的输入信号）+A）·∑/（所有释放 A 的输入信号）

这两个公式的详细描述，大家可以参考《Problems and Solutions in Logic Design》一书[1]里面的第一章“逻辑设计的基本概念”。

下一步向大家介绍如何将时序公式运用到 FSM 的硬件综合里。将用一个例子向大家说明如何设计事件触发 FSM。

回到式（9.1），直接可以得出一个电路，正如图 9.4 所示。

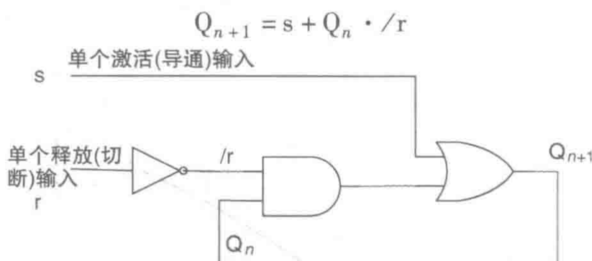


图 9.4　基本事件模块

对于图 9.4 所对应的公式，可以用 De Morgan 法则将其转换成与非的格式：

$$Q_{n+1} = /(/s \cdot /(Q_n \cdot /r)) \tag{9.3}$$

从公式可以看出为什么称它为“与非”时序公式，对应的电路图如图 9.5 所示。

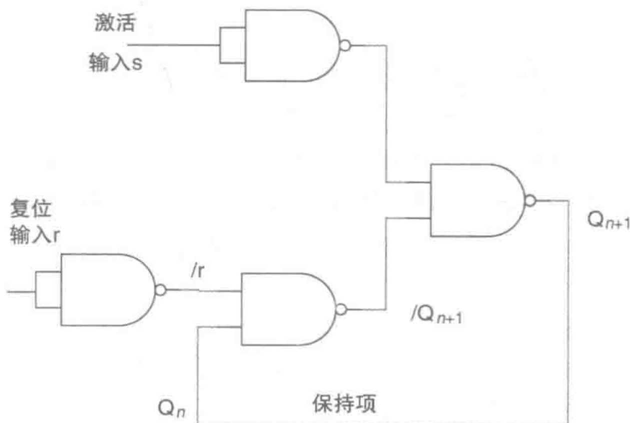


图 9.5　事件模块电路图

两种格式都可以直接应用到设计中去，不过对于 PLD 和 FPGA 器件来说，与门和或门的组合更加方便。

9.3　使用时序公式综合事件 FSM

图 9.6 对应的事件状态图是讨论综合事件触发系统的平台。如何设计事件状态图将在后面的章节详细介绍。系统的基本功能就是判断输入信号 c 端 0 到 1 的状态转换。

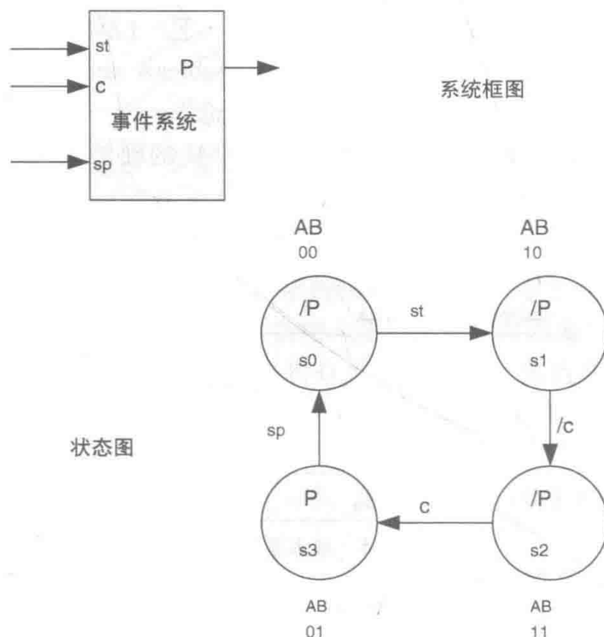


图 9.6　事件触发系统

系统里有三路输入信号 st、c 和 sp。输出信号 P 在状态 s3 的状态是逻辑 1。系统中有两个事件模块：一个是 A，一个是 B。它们同时也是二次状态变量的组成元素。

当输入信号 st 被激活，系统从状态 s0 进入状态 s1，此时系统等待输入 c（输入脉冲）变为逻辑 0（如果 c 在状态 s0 是逻辑 0，FSM 进入状态 s1 之后会直接进入到 s2）。当 FSM 到达状态 s2，系统将等待输入 c 被拉高。这时候，系统就可以捕捉输入 c 从逻辑 0 变为逻辑 1 的状态转换。输出 P 将一直保持逻辑 1 直到输入 s 被激活。这样，可以将 P 看作记录输入 c 状态切换事件的存储单元（或者事件触发指示信号）。

当输入信号 sp 被释放，FSM 将回到状态 s0。

系统可以自动循环，除了可以指示信号 c 逻辑 0 到逻辑 1 的状态切换以外，激活信号 sp 可以让 FSM 返回到初始态，也可以加一个复位输入信号让系统回到 s（这里没有显示）。

首先，事件 A 模块的激活条件必须明确。

$\sum s_A$的建立方法是在系统中寻找所有 A_n从 0 变为 1 所对应的状态。因此，系统中符合条件的状态为 s0 和 s1，在 s0，$A_n=0$，在 s1，$A_n=1$。状态 s0 和 s1 的切换是有条件的，因此需要将 st 纳入公式。所以公式可以写为

$$\sum s_A = s0 \cdot st + s1 \cdot st \tag{9.4}$$

在这里 s1. st 是需要的，原因在于当 FSM 进入状态 s1 之后，输入信号 st 必须仍然保持逻辑 1，目的是确保 FSM 留在状态 s1。

现在：$\sum s_A = /A_n \cdot /B_n \cdot st + A_n . /B_n . st = (/A_n \cdot /B_n + A_n \cdot /B_n) \cdot st = /B_n \cdot st$

公式的化简运用了逻辑结合律。注意：对于事件模块 A_n，最终公式里却没有 A_n了。

然后再看释放条件，对于 A_n，在状态 s2，A_n从逻辑 1 变为逻辑 0。因此

$$\sum r_A = s2 \cdot c + s3 \cdot c \tag{9.5}$$

输入信号 c 在状态 s3 必须保持逻辑 1，目的在于让事件模块保持复位状态。基于状态变量，我们得到$\sum r_A = A_n B_n \cdot c + /A_n \cdot B_n \cdot c = (A_n \cdot B_n + /A_n \cdot B_n) \cdot c = B_n \cdot c$。

这里 A_n又被简化掉了，只剩下 B_n和 c。结果就是事件模块的释放条件：$\sum r_A = B_n \cdot c$

现在可以把完整的时序公式写出来：

$$A_{n+1} = \sum s_A + A_n \cdot \sum /r_A$$

$$A_{n+1} = /B_n \cdot st + A_n \cdot /(B_n \cdot c)$$

上述公式代表了事件模块 A 在系统里的一系列行为模式。时序公式原来的名字叫“与非时序公式”，参考 D. Zissos 教授的《Problems and Solutions in Logic Design》[1]。

事件模块 B 的时序公式可以用同样的办法得出：$B_{n+1} = \sum s_B + B_n \cdot \sum /r_B$

B 的激活条件是

$$\sum s_B = s1 \cdot /c + s2 \cdot /c = A_n \cdot /B_n \cdot /c + A_n \cdot B_n \cdot /c = A_n \cdot /c \tag{9.6}$$

逻辑结合律的应用在这里将 B_n全部抵消，和 A_n在模块 A 的激活条件的情况一样。

B 的释放条件是

$$\sum /r_B = /(s3 \cdot sp + s0 \cdot sp) = /(/A_n \cdot B_n \cdot sp + /A_n \cdot /B_n \cdot sp) = /(/A_n \cdot sp) \tag{9.7}$$

同样地，B_n项被逻辑结合律抵消。因此事件模块 B 的逻辑行为定义已经完成。完整的模块 B 的时序公式为 $B_{n+1} = \sum s_B + B_n \cdot \sum /r_B = A_n \cdot /c + B_n \cdot / (/A_n \cdot sp)$

现在将两个模块的公式归总：

$$A_{n+1} = \sum s_A + A_n \cdot \sum /r_A = /B_n \cdot st + A_n \cdot /(B_n \cdot c)$$

$$B_{n+1} = \sum s_B + B_n \cdot \sum /r_B = A_n \cdot /c + B_n \cdot /(/A_n \cdot sp)$$

代表了两个模块的行为逻辑。

最终的输出信号 P 的公式是基于状态的，好比时钟驱动的系统，本系统中 P 对应状态 s3：

$$P = s3 = /A_n \cdot B_n$$

最后再次强调一下，激活条件（$\sum s_A$）里$/A_n$状态变量被抵消了，释放条件（$\sum /r_A$）里 A_n项被抵消。

同样，$/B_n$和 B_n也在对应的$\sum s_B$和$\sum /r_B$里分别被抵消。

9.3.1 捷径法则

由于在推导公式时逻辑结合律总是要用的，因此在这里有必要再次总结一下状态变量在此过程中的变化。

一种走捷径的判断法则就是，根据逻辑结合律，在事件模块 X 里激活条件$\sum s_x$里是没有/x 项的，而释放条件$\sum /r_x$里是没有 x 项的。再次观察式（9.4）~式（9.7）可以得到验证。

现在公式可以写成$\sum s_A = s0 \cdot st = /A_n \cdot /B_n \cdot st = /B_n \cdot st$

这里将状态变量/A 的 0 到 1 状态转换直接移除，这意味着在式（9.4）里不需要写第二项 s1 · st。

$$\sum r_A = s2 \cdot c = A_n \cdot B_n \cdot c = B_n \cdot c$$

这里将状态变量 A 从 1 变为 0 的转换移除，这意味着在式（9.5）里不再需要第二项 s3 · c 了。

$$\sum s_B = s1 \cdot /c = A_n \cdot /B_n \cdot /c = A_n \cdot /c$$

这里将状态变量/B 从 0 变为 1 的转换移除，因此 s2 · c 这一项在式（9.6）里不再需要了。

$$\sum /r_B = /s3 \cdot sp = /(/A_n \cdot B_n \cdot sp) = /(/A_n \cdot sp)$$

这里将状态变量 B 从 1 变为 0 的转换移除，式（9.7）里 s0 · sp 将不再需要了。

通过上述的简化，向大家提供了一种可以直接从状态图来快速推导时序公式的方法。记住这种方法最简便的途径，就是推导某个状态变量的公式时，直接"丢掉"对应的状态变量。因此，A 模块的公式里，丢掉状态变量/A 在$\sum s_A$里从 0 到 1 变换的部分；在 B 模块的公式里，丢掉状态变量 A 在$\sum r_A$里从 1 到 0 变换的部分。从现在开始，这样的规则将在后面的设计中直接启用。

有了公式以后，它们就可以被运用到 PLD 或者 FPGA 的设计中去了。

9.4 在可编程逻辑器件里运用乘积求和公式的设计方法

这种方法需要将之前公式里的与非部分用乘积项求和来替代。

$$A_{n+1} = \sum s_A + A_n \cdot \sum /r_A = /B_n \cdot st + A_n \cdot /(B_n \cdot c)$$

$$B_{n+1} = \sum s_B + B_n \cdot \sum /r_B = A_n \cdot /c + B_n \cdot /(/A_n \cdot sp)$$

公式 A_{n+1}里，乘积项/（$B_n \cdot c$）可以用 De Morgan 法则来进行拆分：$/(X \cdot Y) = = /X + /Y$，因此$/(B_n \cdot c) = = /B_n + /c$

这样 A_{n+1}公式变为 $A_{n+1} = \sum s_A + A_n \cdot \sum /r_A = /B_n \cdot st + A_n \cdot (/B_n + /c) = /B_n \cdot st + A_n \cdot /B_n + A_n \cdot /c$

同样的公式 B_{n+1}里的/（$/A_n \cdot sp$）可以用 De Morgan 法则变为 $A_n + /sp$。最终公式变为

$B_{n+1} = \sum s_B + B_n \cdot \sum /r_B = A_n \cdot /c + B_n \cdot /(/A_n \cdot sp) = A_n \cdot /c + B_n \cdot (A_n + /sp)$

进一步拆分得到 $B_{n+1} = A_n \cdot /c + A_n \cdot B_n + /sp \cdot B_n$

有了这两个乘积项求和形式的公式，就可以综合事件模块的逻辑电路了。

9.4.1　去掉当前状态和下一个状态的标记：*n* 和 *n*+1

眼下，时序公式的格式一直被写成：$A_{n+1} = \sum s_A + A_n \cdot \sum /r_A$，其中 A_{n+1}代表事件模块的下一个状态。然而，也可以写成：$A = \sum s_A + A \cdot \sum /r_A$，公式左边的 A 代表下一个状态，而右边代表当前状态。这实际上就是一个递归公式。

下面将一直运用这种表示方法。且图 9.7 也很好地说明了这种现象，输出 A 和 B 都被反馈到输入。而且它是系统逻辑电路设计的最终形态，可以直接用 PLD 器件，例如 22V10 进行综合。

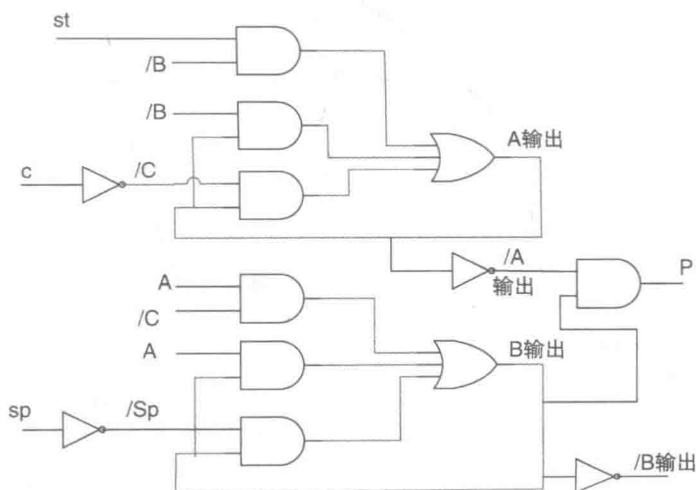


图 9.7　系统最终电路图

9.5 运用事件触发的方法设计带有指示功能的单脉冲发生器 FSM

用时钟驱动的单脉冲发生器在第 1 章就已经向大家介绍过了，这里会稍带复习一下。而且我们的目的是用事件触发模块来再次实现它。

时钟驱动的版本里，输入 p 被激活时（有效），运用系统时钟来控制单脉冲产生的时序。不过，当用事件来触发的情况下，系统里没有时钟，所以需要一个输入（这里命名为 c）来代替时钟控制脉冲的产生（同时它还可以用来设定脉冲宽度）。事件触发系统将在固定的时间间隔内，改变输入信号的状态，让其成为一个“事件”输入。

图 9.8 给出了系统的最终形态。从状态图中可以看到，系统在输入信号 s 被激活后开始工作，但是 FSM 只有在 s 为逻辑 1，c 为逻辑 0 两个条件均满足时，才从状态 s0 进入状态 s1。原因在于信号 c 的值从 0 变到 1 的转换，是用来激活输出信号 P 和 L 的（输出脉冲的开始）。

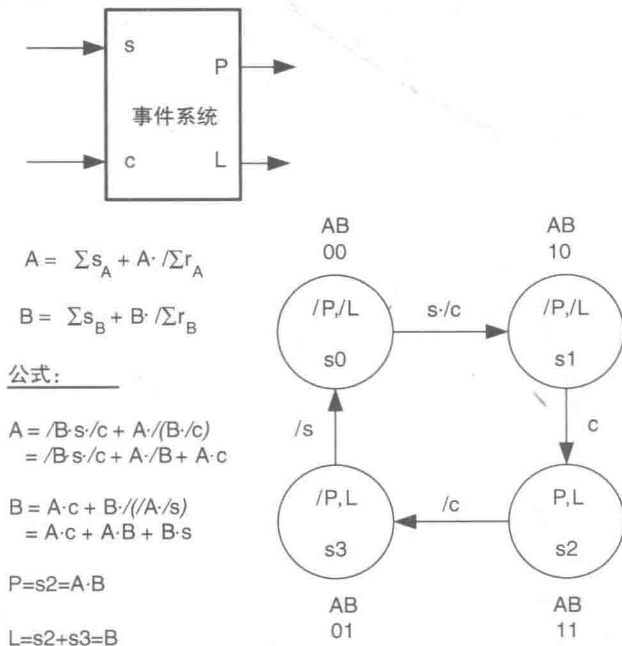


图 9.8 事件触发的单脉冲系统（包括系统框图，状态图和公式）

因此，当 FSM 从状态 s0 转换到 s1 之后，会停在那里等待输入信号 c 变为逻辑 1，然后进入状态 s2，这时 P 和 L 的输出均为逻辑 1。上述情况均在 c 从 0 变为时发生。FSM 将在状态 s2 停留，直到输入 c 重新回到逻辑 0，然后 FSM 会进入状态 s3，此时 P 会被释放，回到逻辑 0，而 L 则仍然保持逻辑 1。

这时候，FSM 会一直处于状态 s3，等待输入信号 s 回到逻辑 0，为下一个脉冲

的产生做好准备。s 回到逻辑 0 后，L 也会被清零。这里的 L 可以看作是脉冲的指示信号，脉冲的宽度是由 c 的脉冲宽度来决定的，而用户不一定能够在界面上看到。

因此在本系统中，P 的实际脉冲宽度是由输入信号 c 在逻辑 1 状态保持的时间长度来决定的。

回到公式上，两个事件模块公式的推导过程可以效仿之前介绍的方法，先得到激活条件、释放条件，然后将其代入时序公式。不过，之前的章节里也介绍了一种便捷的办法，以便可以直接写出系统的公式：

$$A = \sum s_A + A \cdot \sum /r_A = s0 \cdot s/c + A \cdot /(s2 \cdot /c) = /B \cdot s \cdot /c + A \cdot /(B \cdot /c)$$

$$B = \sum s_B + B \cdot \sum /r_B = s1 \cdot c + B \cdot /(s3 \cdot /s) = A \cdot c + B \cdot /(/A \cdot /s)$$

输出的公式为 $P = s2 = A \cdot B$，$L = s2 + s3 = A \cdot B + /A \cdot B = B$。

现在可以将之前的两个事件模块公式转换为乘积项求和的逻辑公式（可直接用于可编程逻辑器件中）：

$$A = /B \cdot s \cdot /c + A \cdot /B + A \cdot c$$

$$B = A \cdot c + B \cdot A + B \cdot s$$

$$P = A \cdot B$$

$$L = B$$

图 9.9 为对应的逻辑电路图。其中的粗线是复位信号，作用是将事件模块初始

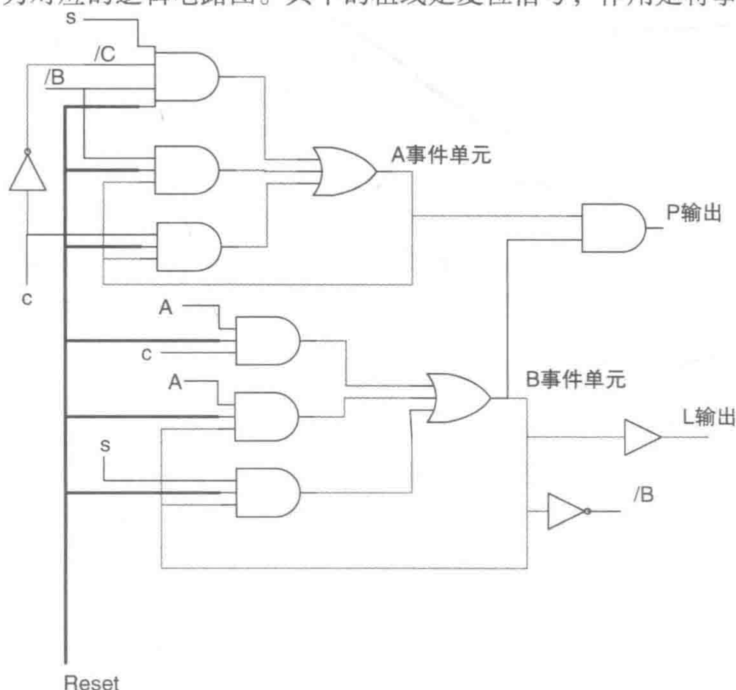


图 9.9　事件触发 FSM 系统逻辑电路图

化到逻辑 0 状态。其重要性在于确保系统可以被强制复位到状态 s0。系统运行过程中，复位信号保持高位（逻辑 1）。一旦触发复位，信号值被拉低，并同时将两个事件模块清零。显而易见，复位信号和事件模块的激活/释放单元是通过与门建立关系的：

$$A = (/B \cdot s \cdot /c + A \cdot /B + A \cdot c) \cdot Reset$$

$$B = (A \cdot c + B \cdot A + B \cdot s) \cdot Reset$$

为了帮助大家分析系统，复位信号没有出现在之前的内容里，但是当大家具体开始设计系统时不能忘记添加进来，否则电路将无法仿真，因为系统不能被初始化。现在，读者可以回到图 9.7 里的例子，试着将复位信号加入到其中，完善一下整个设计。

9.6 另一个事件触发 FSM 的完整案例

本例将完整地阐述事件 FSM 的设计思路，然后用 Verilog HDL 来描述（具体参考第 6 章），最后使用 Syncad™仿真系统进行仿真。

这样做的目的是向大家展示一个完整的设计全过程。Verilog 编写的代码文件将在设计仿真完成后，被烧写到可编程逻辑器件里，用真正的硬件电路来实现设计思路。

9.6.1 重要说明

由于 Verilog 行为级别的描述并没有针对事件触发系统进行优化，到目前为止，所有的描述方式仅限于布尔代数公式层面。这并不影响设计，因为对于系统来说，公式提供了一对一的逻辑关系。同时用逻辑门电路来构建事件触发系统也是可行的。布尔代数公式的特点是比较适用于快速仿真和功能验证。但是基于逻辑门电路的仿真，可以让设计者充分地了解不同类型门电路的延迟，确保系统不会因为 33.3% 的门级响应极限而出现故障（具体原理详见 9.12.3 节和参考文献［1］）。

9.6.2 带有电流监视器的电机控制系统

本节向大家介绍一个用事件 FSM 设计的电机控制系统。系统带有一个监视电机电流的外部器件（可能基于霍尔效应的传感器）。它的工作方式是设定一个固定的电流值，一旦传感器发现电机的工作电流超过预设的值，监视器会发送报警信号给 FSM，让其关掉电机，并点亮报警指示灯（LED）。关于霍尔效应传感器和电机开关电路这两部分在图 9.10a 里并没有体现。

图 9.10b 是 FSM 系统的状态图。输入信号 st 用来控制电机的开关。如果外部传感器检测到过大的电流，其输出信号 ms 将被拉高，此时 FSM 将进入状态 s2，电机将会被关闭，且指示灯 L 将点亮（低有效）。系统将停在状态 s2 直到输入信号

被释放，随后系统将进入状态 s3，将告警指示灯 L 熄灭。输入信号 t 如果在状态机处于状态 s3 时被释放，可以让其回到初始态 s0。

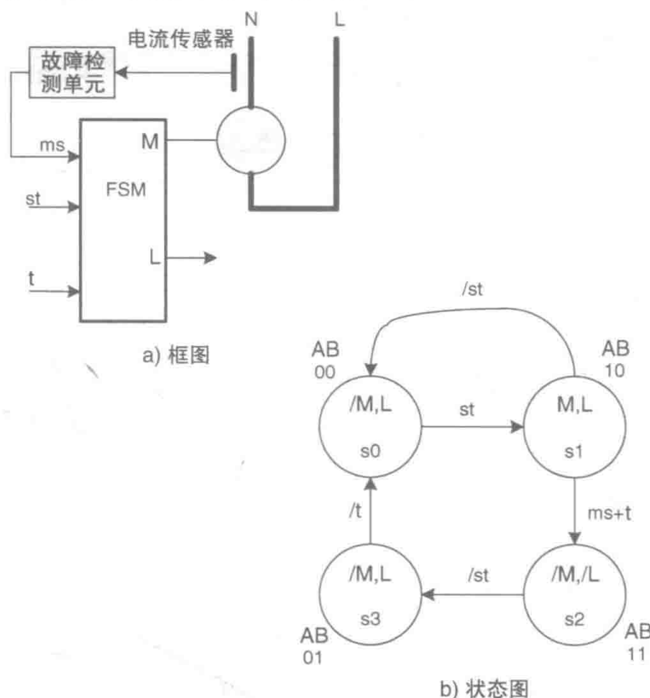


图 9.10　电机控制器的系统框图和状态图

系统在没有收到报警信号的情况下，通过控制输入信号 t 仍然可以进行功能测试。要注意的是 t = 1 的目的是让 FSM 停留在状态 s3。事件模块的公式推导过程如下：

$$
\begin{aligned}
A &= \sum s_A + A \cdot \sum /r_A \\
&= s0 \cdot st + A \cdot /(s1 \cdot /st + s2 \cdot /st) \\
&= /B \cdot st + A \cdot /(/B \cdot /st + B \cdot /st) \\
&= /B \cdot st + A \cdot //st \\
&= /B \cdot st + A \cdot st
\end{aligned}
$$

$$
\begin{aligned}
B &= \sum s_B + B \cdot \sum /r_B \\
&= s1 \cdot (ms + t) + B \cdot /(s3 \cdot /t) \\
&= A \cdot ms + A \cdot t + B \cdot /(/A \cdot /t) \\
&= A \cdot ms + A \cdot t + A \cdot B + B \cdot t
\end{aligned}
$$

$$M = s1 = A \cdot /B$$

$$L = /s2 = /(A \cdot B) = /A + /B$$

图 9.11 是电路图，图里已经带有了测试信号 t，所以状态机在没有收到报警信号的

情况下仍然可以返回初始态。

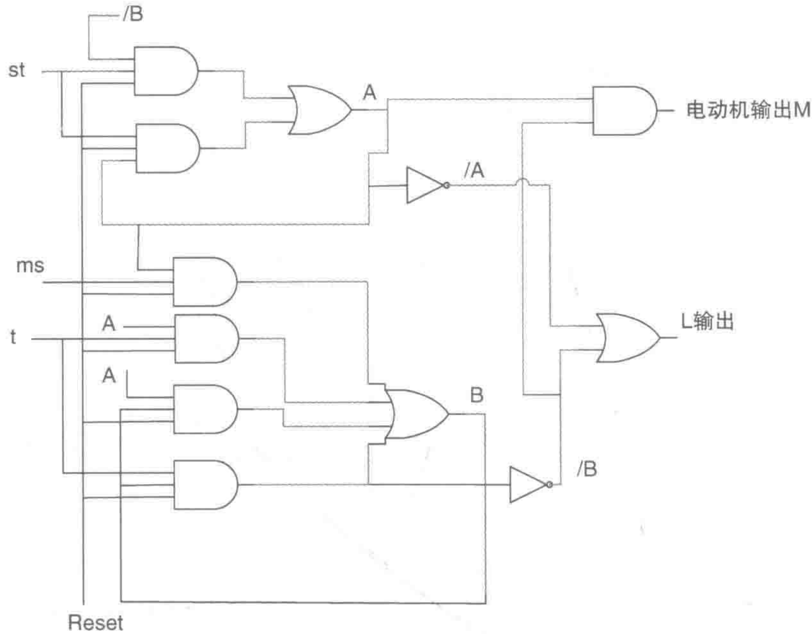


图 9.11 状态机电路图

尽管画出电路图的必要性不大，但是有 FSM 的电路图更加易于帮助理解。要注意的是，FSM 是低电压数字电路，而电机有可能是高压电路驱动，这之间信号传递所需要的基本电压转换和缓冲系统并没有给出。

用 Verilog 语言模块来实现的结果，如代码 9.1 所示。

```
///////////////////////////////////////////////////
module fsm (rst, st, ms, t, M, L, A, B);
  output M, L, A, B;
  input rst, st, sp, ms, t;

assign
  A= (~B &st | A &st) &rst,
  B= (A &ms | A & t | A & B | B & t) &rst,
  M=A & ~B,
  L= ~A |  ~B;
endmodule
///////////////////////////////////////////////////
```

代码 9.1 FSM 语言模块

模块的输入和输出是在标题的括号外面定义的，和老版本的 Verilog 格式一样对于新版本的编译工具同样适用。关于如何定义输入和输出请大家参考第 6 章，

更详细的介绍。

Verilog 代码中的事件公式使用了阻塞语句，并且加入了测试信号 t，模拟系统没有遇到报警的场合。

代码 9.2 用于仿真测试平台。测试平台模拟了 FSM 的环境，并加入一系列的测试信号验证功能是否完善。

```
module test;
reg st, ms, t, rst;

fsm uut (rst, st, ms, t, M, L, A, B);

initial
begin
    $dumpfile (“motflt.vcd”); //为了获得打印的输出波形
    $dumpvars;
   Rst =0;
   St =0;
   Ms =0;
   T =0;
//需要注意的是确保所有信号的变化是按照一定顺序的。
//并且确保信号 ms 和 st 是相互独立的。
//-------------------------------释放复位信号
   #20rst =1;
//-------------------------------进入状态 s1
   #20 st =1;
//-------------------------------停在状态 s1
//-------------------------------进入状态 s0
   #20 st =0;
//-------------------------------再次进入状态 s1
   #20 st =1;
//------------------------------进入状态 s2
   #20 ms =1;
//------------------------------
   #30 ms =0;
//------------------------------进入状态 s3
   #20 st =0;
//------------------------------回到状态 s0
   #20 st =0;
//------------------------------进入状态 s1
   #20 st =1;
```

```
//-------------------------------------停在状态 s1
//-------------------------------------进入状态 s2
   #20 t =1;
//-------------------------------------进入状态 s3
   #20 st =0;
//-------------------------------------进入状态 s0
//-------------------------------------测试结束

$stop (60); //结束仿真
end
endmodule
```

代码 9.2　测试平台代码

FSM 模块的内部构成比较直观，除了输入和输出以外，只有一个赋值语句模块。事件模块和输出公式的定义都在同一个模块内完成。

测试平台的代码也很简单。有一点必须强调的是，为了避免潜在的静态 0 或 1 冒险（错误状态），某一时刻只能有一个信号发生变化，并需要延迟一段时间。这也是设计事件触发系统的一个必要条件。

最后，图 9.12 是仿真的波形图。

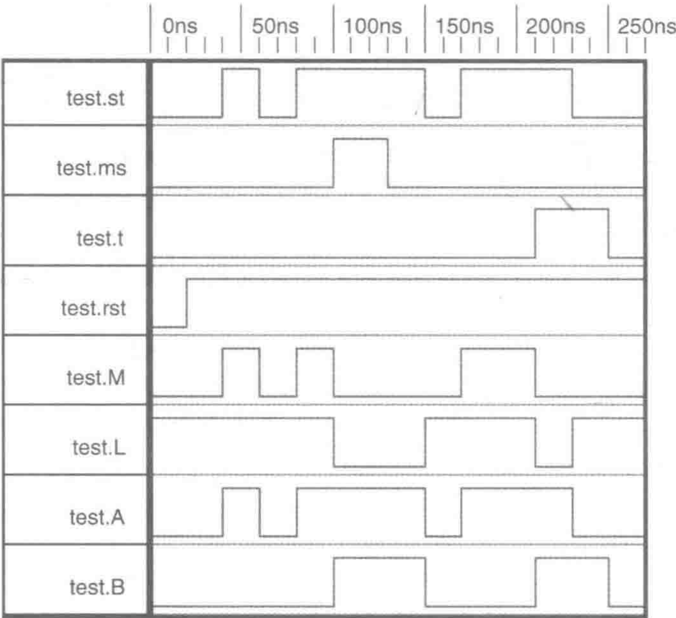


图 9.12　系统 Verilog 仿真图

将波形图和仿真代码进行对照，可以发现 FSM 进行了两次循环，一次带有报警信号，另一次带有测试信号 t。

当系统有报警时，系统从状态 s2 进入状态 s3，再回到 s0 这个过程是非常迅速的，因此从仿真图上观察状态 s3 的停留不是十分明显。当仿真系统接收测试信号时，FSM 会停在状态 s3 直到输入信号 t 回到逻辑 0。通过这样的方法，系统的测试覆盖了 FSM 所有可能的情况（特别是可以用状态变量来表达 LED 指示灯输出信号的情况下）。

9.7　用 FSM 控制悬停式割草机

悬停式割草机通常使用一个机械的内部锁止结构，来防止电机在操作员没有按下解锁按钮的情况下自动误操作。如果用电子系统来代替原来的机械结构，整个割草机的安全操作系统在流水线生产的过程中，装配会更加便捷。

9.7.1　系统描述和解决方案

割草机系统内部有一个解锁按钮 sf，其目的是在启动操作杆 st 之前这个按钮必须被按下。当这个锁止按钮被按下后，LED 指示灯 P 会被点亮；随后如果启动操作杆 st，电机将开始工作。如果释放操作杆，电机将停止工作。解锁按钮在电机重新投入工作之前必须再次被按下。

图 9.13a 和 b 是描述上述过程对应的系统框图和状态图。之前的描述也可作为产品功能描述的一部分出现在说明书里。对照原有的状态图（图 9.13b 里 4 个状态的版本），一系列安全措施已经被加入到系统中。这也是在整个系统控制流程已经明确的基础上，设计 FSM 过程中逐步形成的。

整个流程的控制是建立在确保锁止按钮，在操作杆被启动之前按下，这个前提上完成的。如果操作杆启动信号 st 被释放，LED 指示灯 P 在状态 s1 仍然将保持被点亮的状态。如果锁止按钮一旦被释放，不管 FSM 在 s1 还是 s2，都将立刻回到状态 s0。

要注意的是，操作杆或者锁止按钮任何一个被释放，FSM 都将首先回到状态 s1。这种设计确保了只有当操作员的双手一起操作两个控制端，割草机的电机才能工作。操作员必须看到 LED 指示灯 P 被点亮后，才能控制操作杆启动电机。最后要注意没有实际功能的状态 s3 会被引导回状态 s0。这样确保如果系统因为出现一些故障进入 s3，它将立刻回到状态 s0。

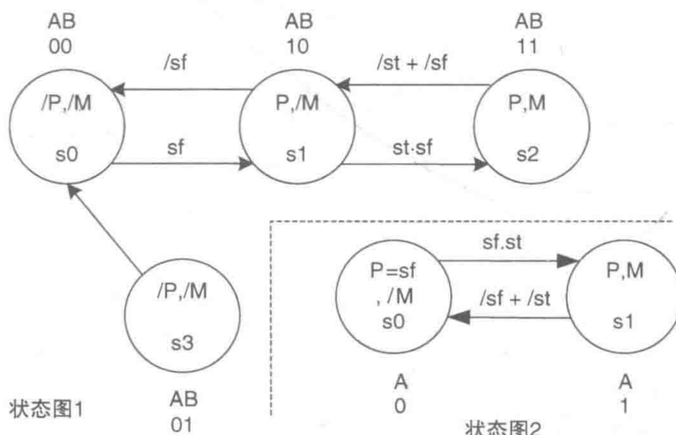
图 9.13b 的另一个状态图（只有两个状态）提供了第二种解决方案，这里在输入端运用组合逻辑（如状态传输线上所示）。逻辑公式可以用常规的方法来推导：

$$
\begin{aligned}
A &= \sum s_A + A \cdot \sum /r_A \\
&= s0 \cdot st \cdot sf + A \cdot /(s1 \cdot (/st + /sf)) \\
&= st \cdot sf + A \cdot /(/st + /sf) \\
&= st \cdot sf + A \cdot //(st \cdot sf) \\
&= st \cdot sf + A \cdot st \cdot sf
\end{aligned}
$$

$$P = s0 \cdot sf = sf$$



a)



b)

图 9.13　a）割草机状态机控制框图 b）两种状态图解决方案

从 P 的公式里可看出指示灯在状态 s0 时就可以被点亮（米利型输出），同样在状态 s1 也是，条件是 sf · st。

$$M = s1 = A$$

对于系统来说公式被大幅简化后，只需要 3 个逻辑门：两个与门和一个或门，而输出信号 P 和 M 是需要缓冲器来过渡的。

图 9.14 的电路图给出的是图 9.13 的第二个状态图所对应的情况。外加的缓冲器用来确保电压的稳定。特别要说明的是，电机输出信号 M 需要连接一个继电器（静态的或者机电的），用来将 FSM 和割草机的电源有效隔离。

状态图 2 使用 Verilog 的逻辑门语言进行描述。这样设计者可以在单个逻辑门之间增加必要的传输延迟。如代码 9.3 所示：

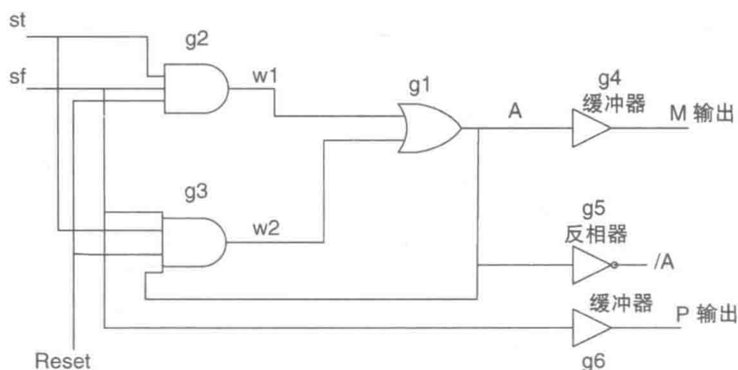


图 9.14　割草机的 FSM 对应电路图

```
module mowerfsm (st, sf, P, M, A, rst);

input st, sf, rst;
output P, M, A;
wire na, nb, w1, w2;

or #5 g1 (A, w1, w2);
and #5 g2 (w1, sf, st, rst);
and #5 g3 (w2, A, st, sf, rst);
  //-----------------------------
buf #5 g4 (M, A);
  //-----------------------------
not #5 g5 (na, A);
buf #5 g6 (P, sf);
  //----------------------------
endmodule
```

代码 9.3　割草机模块代码

测试模块代码参见代码 9.4：

```
module test;
reg rst, st, sf;
mowerfsmuut (st, sf, P, M, A, rst);

initial
    begin
      $dumpfile ( "mower.vcd");
      $dumpvars;
    rst =0;
    st =0;
```

```
    sf =0;
      #20 rst =1;
      #20 sf =1;
      #20
      #20 st =1;
      #20
      #20 st =0;
      #20
      #20 st =1;
      #20
      #20 sf =0;
      #20
      #20 st =1;
      #20
      #20 st =0;
      #10  $finish;
    end
endmodule
```

代码 9.4 测试固件模块

图 9.15 所示的仿真图是基于图 9.13 中的第二个状态图的结果，且仿真过程是按照代码 9.4 里的代码顺序来的。

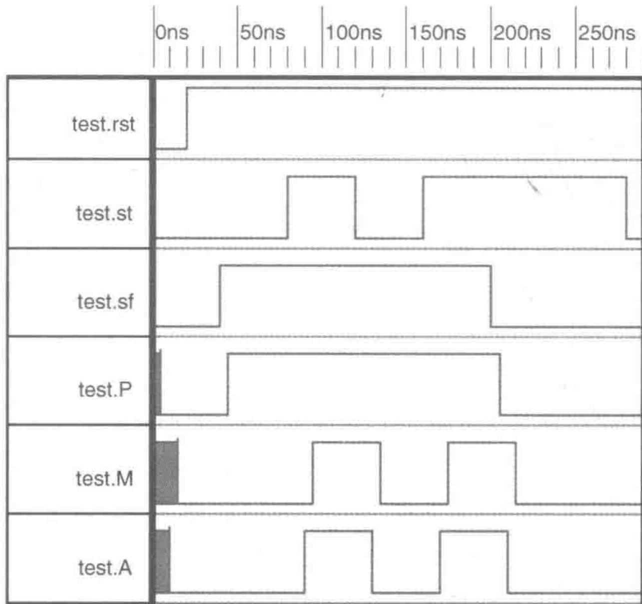


图 9.15 割草机仿真结果

仿真从激活 sf 输入信号开始，然后指示灯 P 被点亮。随后输入信号 st 被拉高，割草机电机开始运转。输入信号 st 被释放后，电机停止。这里可以反复开启或者暂停电机，只要信号 sf 处于激活状态。随后信号 sf 被释放（此时开始信号 st 仍然有效），电机被关闭。

现在再回顾一下系统的设计，仔细思考可以发现割草机的控制，可以简化成一个组合逻辑：$M = sf \cdot st$ 和 $P = sf$。

这个结果现在通过上述一系列分析变得显而易见，可能有读者在一开始就意识到最终结果是这样。简化后的公式等同于纯机械结构的开关设计。

图 9.13 里第一个状态图是原本的设计方案，它虽然也没错，但是需要 3 个状态和 2 个事件模块。而第二个方案在同样解决问题的前提下使用了更少的状态。最终，组合逻辑的方案提供了最简洁的设计思路。解决问题的同时需要观察它是否能够被最大幅度地简化，需要花费不少的精力。用时序模块来设计的方案，很容易导致过度消耗资源。

9.8　没有输入条件的状态切换

图 9.16 所示的 FSM 里，从状态 s3 到状态 s0 之间没有任何输入条件。

FSM 公式如下：

$$A = \sum s_A + A \cdot \sum /r_A$$
$$= s0 \cdot m + A \cdot /(s2 \cdot p)$$
$$= /B \cdot m + A \cdot /(B \cdot p)$$
$$= /B \cdot m + A/B + A/p$$

状态变量 B 的公式将不用之前提到的快速推导的捷径：

$$B = \sum s_B + B \cdot \sum /r_B$$
$$= s1 \cdot /m + s2 \cdot /m + B \cdot /(s3 + s0)$$
$$= A \cdot /B \cdot /m + A \cdot B \cdot /m + B \cdot /(/A \cdot B + /A \cdot /B)$$
$$= A \cdot /m + B \cdot /(/A)$$
$$= A \cdot /m + B \cdot A$$

在状态变量 B 的公式里，$\sum /r_A$ 项（通过捷径省略）就是//s3，也就是//A，因为传输线上没有输入条件。

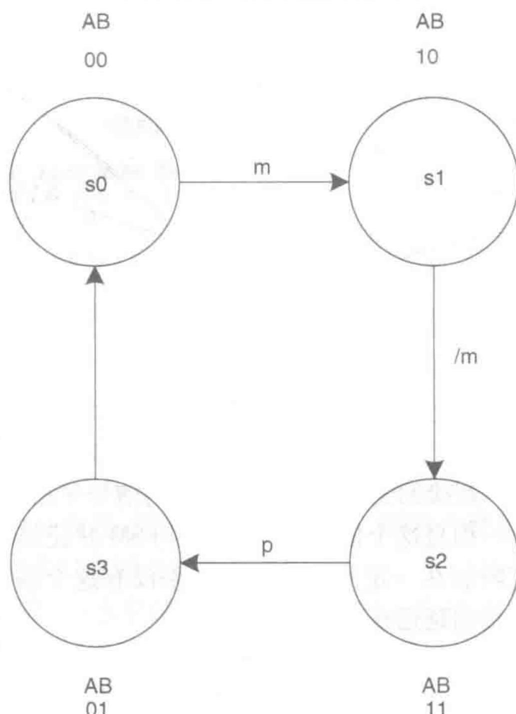


图 9.16　没有输入条件的状态图

这个 FSM 没有任何输出（大部分 FSM 应该是有的），因此这只是一个学

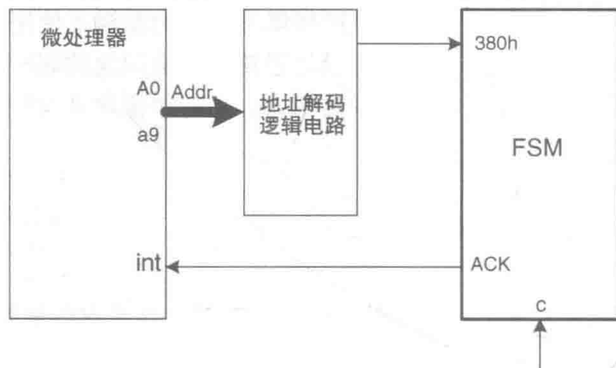
术性较强的示例，没有特别大的实际意义。

记住，如果要仿真，务必加上复位信号。

9.9　特例：微处理器地址空间响应

现在看一个比较特殊的例子。假设有一个基于 FSM 的事件控制芯片（PLD 或 FPGA），需要和一个微处理器进行同步。下面来讨论相应的解决方案。

如图 9.17 所示的系统，地址 380h 是由地址解码逻辑电路产生的。这类电路可能在 PLD 或 FPGA 内部生成。而输出结果就是信号 380h，它被用来控制 FSM。



时钟是作为一个事件输入

地址380hex是如何构成的:

a9 a8　a7 a6 a5 a4　a3 a2 a1 a0
1 1　1 0 0 0　0 0 0 0
3　8　0

图 9.17　基本的地址激活系统框图

当信号 c 为逻辑 0 时，FSM 将对此地址进行响应并进入状态 s1，然后等待信号 c 被拉高。

信号 c 被拉高后，FSM 将进入状态 s2，并截获 ACK 信号，"告诉"微处理器它已经"看到"了 380h 信号。当信号 c 再次回到逻辑 0 之后，FSM 将从 s2 经过 s3，直接回到其初始态。这里的信号 c 相当于系统时钟。

而且这个信号是用来控制 FSM 并让其回到初始态，并且可以将信号 ACK 的脉宽限制在一定的范围内。如果没有这个信号 c，那么 ACK 的脉宽将由逻辑门之间的传输延迟来决定。

图 9.18 是 FSM 的框图。图中所带的激活和释放条件是由地址解码器和时钟信号 c 产生。因此这是一个比较简单的事件触发 FSM，它提供了一种不需要太多逻辑电路就能产生控制信号的方法。

在微处理器系统里，地址解码电路很多时候是现成的；而在微控制器系统中

FPGA 可以在片内集成解码逻辑电路和 FSM，尽管这样设计比较消耗片上资源。图 9.17 中的 ACK 信号可以用来产生一个中断信号，这样避免占用一个芯片的管脚作为输入信号。

上述系统想要正常工作有一个条件，就是微处理器和 FSM 之间的时钟信号 c 的布线要足够短，这样可以避免受到传输线延迟的影响。还有一个条件就是时钟周期要远大于 FSM 里的传输延迟，这样以便于让 FSM 有缓冲的时间。

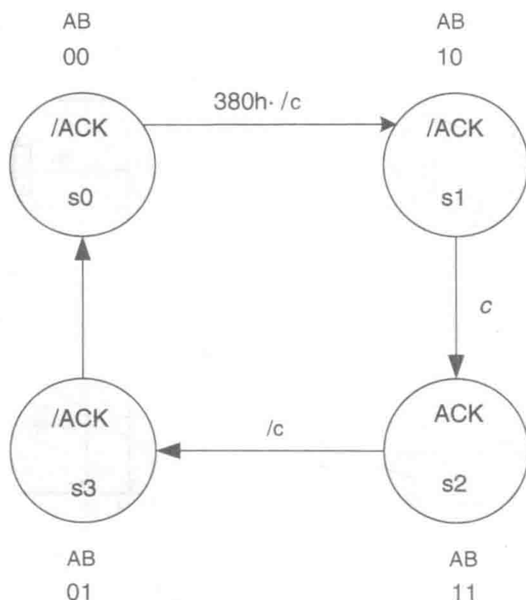


图 9.18 地址激活状态机的状态图

公式如下：

$$
\begin{aligned}
A &= \sum s_A + A \cdot \sum /r_A \\
&= s0 \cdot 380h \cdot /c + A \cdot /(s2 \cdot /c) \\
&= /B \cdot 380h \cdot /c + A \cdot /(B \cdot /c) \\
B &= \sum s_B + B \cdot \sum /r_B \\
&= s1 \cdot c + B \cdot /s3 \\
&= A \cdot c + B \cdot //A \\
&= A \cdot c + B \cdot A \\
ACK &= s2 \\
&= A \cdot B
\end{aligned}
$$

9.10 运用米利（Mealy）型输出

有时候一个输出和一个或者多个输入之间存在函数关系，不过只是在某些特定的状态才出现。大家可以回顾一下最初学习米利型 FSM 时，一些外部输入信号同时被连接到外部解码单元。下面通过一个例子来具体说明。

9.10.1 水箱水位控制系统的解决方案

图 9.19 使用了一个水泵向水箱注水（通过将信号 P1 置 1，P2 置 0）。目的是为了将水箱加注到规定的水位，如果水位处在传感器 Sh 和 Sl 之间，则符合要求。水位一旦符合要求了，水箱的出水口和进水口之间会被水泵平衡。

如果水位降到 Sl 以下（l1 信号被激活），水泵进入高速运转状态，此刻 P1 = 0，P2 = 1。这样就避免了在水箱的出水口一端出现空气阻力现象。

等水位升到 Sh（l2 信号被激活），水泵将被关闭。

系统为了保持水位会持续运转。当然它可以通过信号 st 和 sp 被开启和关闭，如果需要可以将开关信号由 2 个变为 1 个。

表 9.3 给出了水位传感器输入信号 l1 和 l2 的关系，以及它们如何影响水泵 P1 和 P2 的运转情况。注意到表格的最后一行是根据实际情况推断的。很明显，不可能出现水箱没水，高水位指示信号却又有效的情况。

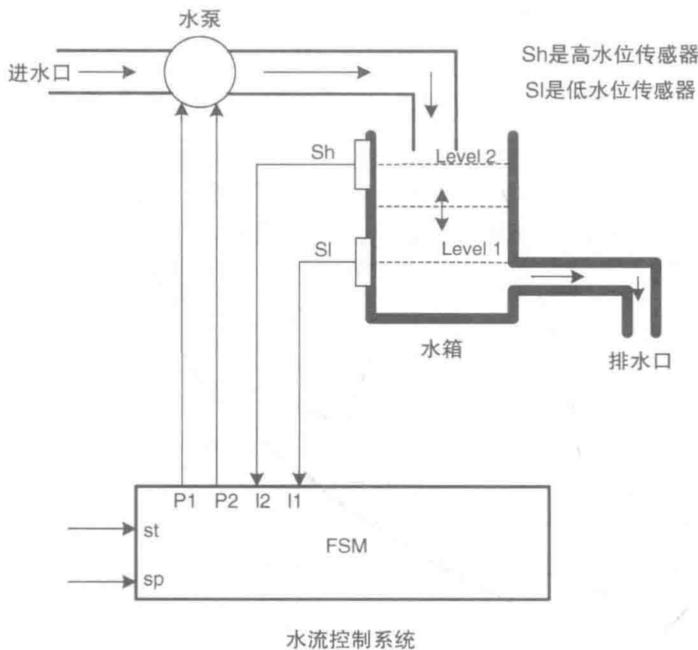


图 9.19　基于米利型 FSM 的水泵控制系统框图

表 9.3　水位传感器和水泵输出信号之间的关系表

l1	l2	P1	P2	注释
1	1	0	0	水泵关闭；防止水位过高而溢出
1	0	1	0	水泵正常工作；水位在两个传感器之间
0	0	0	1	水泵加速运转；水位低于 Sl 传感器
0	1	0	0	不存在的情况；水泵关闭

由此，可以设计出和上面描述一致的状态图，如图 9.20 所示。图中，系统复位到待机状态，并等待开始信号。一旦检测到开始信号，系统进入状态 s1，即休眠状态。只要水位保持在 l1 传感器之上，系统都将停留在 s1。当 FSM 即将进入状态 s3 时，水位传感器将开始主导系统的运行。这种情况只有在传感器 l1 为 0 时出现，因此信号 P2 可以将水泵加速，让水位回到低位以上。一旦系统进入状态 s3，它将在状态 s3 和状态 s2 之间来回切换以保证正常的水位。

系统可以在任何时候停止工作并回到状态 s0。如果在状态 s3 出现停止信号，P2 将被强行拉低。这样就避免了水泵的转速从状态 s2 到 s3 到 s1 再到 s0 的过程中，出现从正常到高速再到停止的非正常运转。一旦系统被关闭，水位将跌落到

箱清空的状态。如果系统在打开时水箱是空的，状态机将从 s0 进入 s1，随后直接进入 s3 快速加注水箱，让水位处于 l1 和 l2 之间。

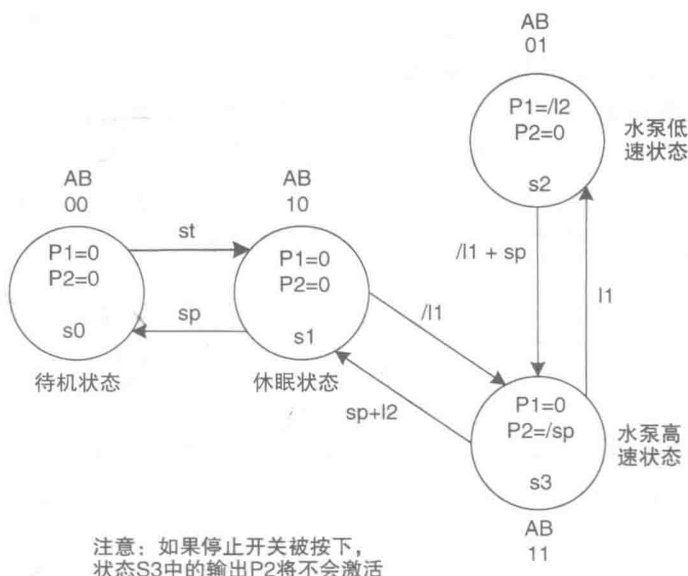


图 9.20 带米利输出的 FSM 初步解决方案

如果将二次状态变量加入状态图，就可以获得一个比较实用的系统解决方案。其中，二次状态赋值可以是 s0 = /A/B，s1 = A/B，s2 = /AB，s3 = AB；

或者 s0 = /A/B，s1 = /AB，s2 = A/B，s3 = AB。

可能读者会思考是否可以将这个 FSM 进行简化。事实上，对照系统里的输入和输出关系，可以得出一系列组合逻辑关系方程。这是因为水流的持续流动形成了输入和输出信号之间自然有序的变化。回顾表 9.3 里最后一行对应的 l1 为逻辑 0，而 l2 为逻辑 1 的情况，水泵必须保持关闭。P1 和 P2 的关系方程为：P1 = l1 · /l2，P2 = /l1 · /l2。

然而，这样写是不行的，因为要考虑到开始和关闭信号也在系统里起作用。假设这两个开关是按钮，系统需要配置存储模块来将开关纳入公式。

图 9.21 是最终的系统解决方案，这时的系统只有在状态 s1 才运转，在 s0 是不工作的。

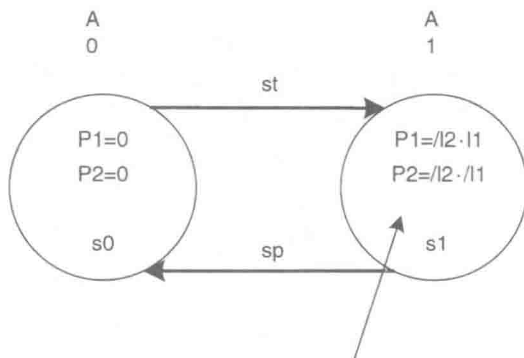


图 9.21 最终方案：米利输出 FSM

关于 P1 和 P2 的公式只有在 FSM 处于状态 s1 时才有效。因此，这两个输入信号的公式应该写成：P1 = s1 · /l2 · l1，P2 = s1 · /l2 · /l1。

为了获得事件模块（这里只有一个事件模块）：$A = \sum s_A + A \cdot \sum /r_A$，因此：A = s0 · st + A · /（s1 · sp）。

将 s0 和 s1 用二次状态变量替代后得到：A = /A · st + A · /（A · sp）

/A · st 里的/A 和 A · sp 里的 A 需要被抵消，剩下就是：A = st + A · /sp。

当我们抵消某一项时，例如/A · st，此时/A 变为 1，所以实际上最终变为 1 · st，那么/A · 1 = /A。

同样地，A · sp 变为 1 · A · sp，去掉 A 之后就是 1 · sp = sp。所以最终系统各项公式汇总为：

A = st + A · /sp

P1 = A · l1 · /l2

P2 = A · /l1 · /l2

最后，还可以将这个例子看作组合逻辑电路来设计，而不需要事件模块。鉴于这个例子的特殊性，是完全可以这么做的。水位传感器是根据水箱里的水量来进行时序操作的。

P1 = l1 · /l2 · st · /sp

P2 = /l1 · /l2 · st · /sp

上述公式只有当系统中使用的开关在释放后保持打开或者关闭状态的情况下才能成立。如果系统使用那种状态随按下或者抬起而变化的按钮（即按下激活，抬起释放），则需要一个事件模块来“记住”开关上所做的动作。

9.11 使用继电器的电路

事件时序公式也可以加入继电器模块来进行系统设计。如今使用继电器设计 FSM 可能显得有些过时，但是，在一些特殊的情况下，老式机电结构的继电器更加可取。可以替代的方法是用半导体静电继电器，来取代传统的机电继电器。这两者都可以用在高电压或者高电流的工作环境下。

下面要介绍的例子中带有逻辑门和继电器电路。

请大家看下面的系统描述，这和上面章节 9.6 里的电机控制问题比较类似。

一个带有开始和停止开关的电机，按下开始按钮电机就会运转，此时停止按钮没有被按下。如果要关掉电机，按下停止按钮，开始按钮此时处于未被按下的状态。如果停止按钮按下时，开始按钮也处于被按下的状态，那么电机即将停止工作，并且会有一个 LED 指示灯被点亮。系统只有在操作员按下复位按钮时才能回到初始态。复位按钮同时也可以用来释放系统的当前状态，不管开始和停止按钮哪个被按下。

图 9.22 所示的状态图和上述系统描述相对应。在开始按钮 st 被按下后，电机开始工作，FSM 从 s0 进入 s1，但是限制条件是停止按钮 sp = 0。FSM 在 s1 时，如果此刻操作员按下停止按钮，它会回到状态 s0，且此时 st = 0。按下停止按钮时，如果开始按钮也处于激活状态，会让 FSM 进入状态 s2，状态 s2 是一个锁死状态（这意味着如果不进行强行复位，FSM 会一直停留在 s2）。这样设计的目的是让复位输入信号将 FSM 拉回初始态（s0）。

现在，可以将公式推导出来：

$$A = /B \cdot st \cdot /sp + A \cdot B + A \cdot /sp + A \cdot st$$

$$B = A \cdot st \cdot sp + B \cdot /0$$

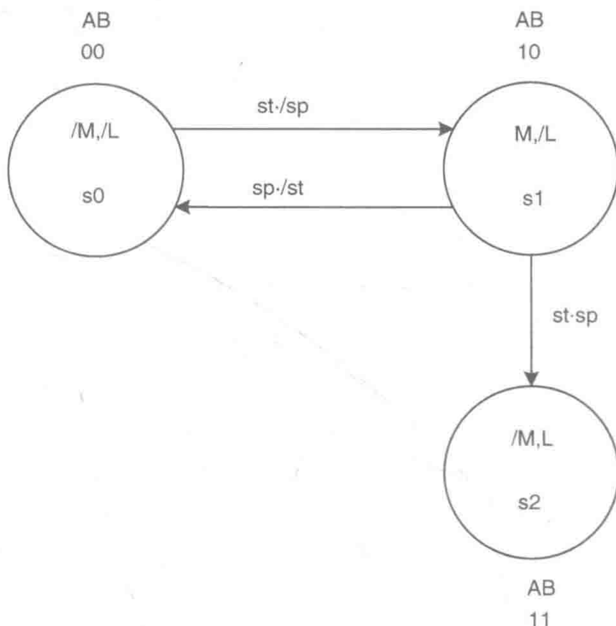


图 9.22　电机控制状态图

注意 B 的公式里没有释放条件，因此取反项是/0，也就是逻辑 1。

输出的公式为：$M = s1 = A \cdot /B$，$L = s2 = A \cdot B$。

基于上述公式，可以用 PLD 器件，也可以用继电器来设计电路。

图 9.23 使用与门、或门和非门来构建逻辑原理图，因此它适合用 PLD 器件来综合。注意到事件模块 B 需要反馈回来和复位信号一起作为与门的两个输入端，这样才能将模块复位到初始态。

不过，不难想象这样的电路是需要低电压 5V 来供电的，并且还需要从电机的

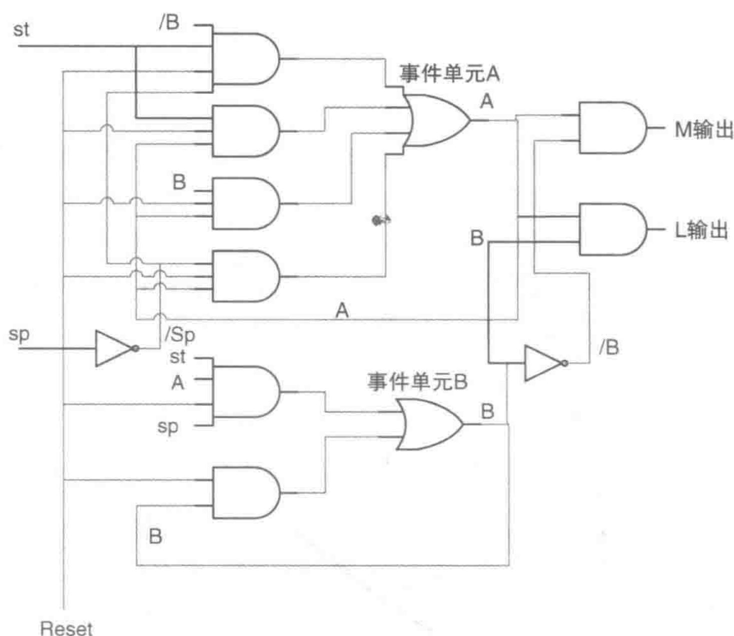


图 9.23 电机控制逻辑电路图

主电源取电，这样就不得不加入一个变压器和整流器来把电压降下来。变压器可以用降压电阻、二极管和电容所组成的电路来完成，但这些仍然需要一定的费用。

另一种思路就是使用基于机电继电器或者静电继电器的方法。这种方法的好处就是可以直接承载主电源较高的电压（当然继电器的选型必须符合要求）。继电器电路可以直接从时序公式获得。

图 9.24 是基于继电器的解决方案。图中，继电器触点的位置表明继电器不在工作状态。电路使用一个简单的半波整流器和相应的电容串联，来获得继电器 A 和 B 所需的直流电压。电容旁边与之并联的电阻在系统切断电源（或者复位）时为其提供放电通道。

图中两个圆圈代表了继电器 A 和 B 的线圈（或者是静电继电器的输入端）。与线圈并联的二极管用于当继电器开关断开后，承载瞬间电压造成的电流冲击；否则，经过线圈的 EMF（感应电动势）将有可能损坏开关和继电器触点电路。这些二极管通常被称为“箝位二极管”。

在进一步学习异步（事件触发）FSM 之前，需要事先搞明白竞争冒险效应对于事件触发 FSM 所带来的影响。

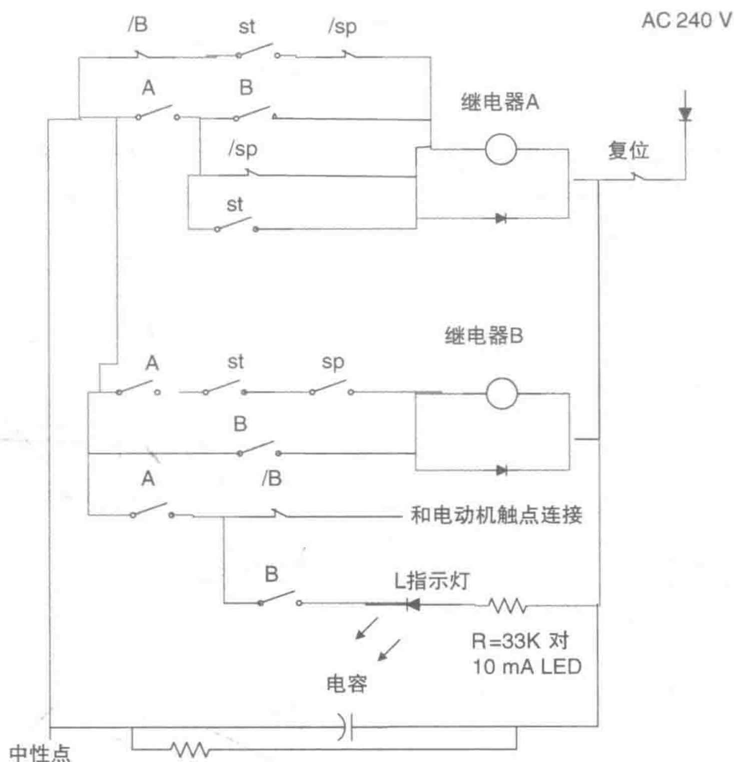


图 9.24　基于继电器的电机控制系统

9.12　事件触发 FSM 里竞争冒险的条件

本节将对异步 FSM 的设计过程中，可能遇到的一些问题进行剖析，并讨论相应的解决方案。

对于事件 FSM，产生竞争冒险的潜在条件有以下 3 种：

1）输入信号之间的竞争；

2）二次状态变量之间的竞争（事件模块本身）；

3）主要变量和二次变量之间的竞争。

这对于 FSM 的设计至关重要，因为对于事件触发系统，存在这类潜在的竞争冒险会导致整个系统的设计出现错误。关于本节的内容也可以参考牛津大学出版社的《Problems and Solutions in Logic Design》[1]。

9.12.1　输入信号之间的竞争

这种情况一般出现在当 FSM 面对一个三路状态分支的情况，当需要其进入其中一个状态时，影响其移动方向的两个输入信号同时发生变化。很明显，一般情况下是无法保证两个（或多个）输入信号在同一个时间点发生变化，因为信号线传

输过程中的延迟始终存在。

注意：为了避免这种竞争，在设计时尽量避免让两个（或多个）信号同时发生变化。

图 9.18 中状态变化条件就有两个，即 380h · /c，但是这里 FSM 进入下一个状态的条件是 380h 和/c 相与，而且在进入下一个状态之前，c 的值即将被拉高，这里还有一个前提就是 FSM 到达当前状态的条件是 c 的值为低。所以，这和一般情况不一样，这里的输入信号有固定的变化方向，如果在同一时刻变化是不会造成混淆的。

9.12.2 二次状态变量之间的竞争

这种情况一般发生在设计者没有在 FSM 的二次变量里运用单位距离编码（例如 A、B 是事件模块）。这种不使用单位距离编码的 FSM 有可能进入某个未知状态，导致事件模块之间不对等的传输延迟。

让我们再看下图 9.18 里的例子，其中二次状态变量的表达式为：s0 = /A · /B，s1 = A · B，s2 = A · /B，s3 = /A · B。

如果在状态 s0，380h 输入端是逻辑 1，c 是逻辑 0，那么 A 此时在 B 之前发生变化，会导致 FSM 从 s0 突然跳到 s2。在状态 s2 时，因为 c 仍然是逻辑 0，下一个状态是 s3。因为图中 s3 和 s0 之间没有输入条件，所以 FSM 会直接回到 s0。这种情况是没有办法预见的，因为如果在 s0 时是 B 首先变为逻辑 1，则 FSM 会直接从 s0 到 s3，再到 s0。

记住，在异步（事件触发）FSM 里没有同步时钟，因此就不会产生让信号的变化稳定下来的延迟。

解决办法：在设计异步 FSM 时，尽量使用单位距离编码来定义二次状态变量。

9.12.3 主要变量和二次变量之间的竞争

最后一种竞争也是最复杂的一种情况，大家可以对照参考文献［1］来获得更多的信息。

基本上，就像标题所说的，这是在主要（外部输入）变量和事件模块（二次状态变量）之间的竞争。导致竞争的原因是：当外部输入对模块产生影响的传输延迟大于事件模块内部变化时传输过程中的延迟（模块输出到模块输入导致模块被激活或者释放）。此时会导致事件模块的误操作。

为了避免这种竞争的发生，就要确保主延迟 Tp 永远小于二次延迟 Ts，即 Tp < Ts，确切地说为 $Tp_{max} < Ts_{min}$。

这和参考文献［1］里的概念是一致的，这里再次引用一下原书的定义 $Tp_{max}/Ts_{min} < 1$。

这里 Tp_{max} 是最大可能的主要输入信号的路径延迟，而 Ts_{min} 是最小可能的二次

状态变量变化时的路径延迟（例如 A 和 B 输出所对应的所有逻辑门延迟）。

本书中所涉及的异步 FSM 里的事件模块之间门电路延迟的最大允许误差范围在 33.3% 之内，符合上述避免竞争的条件。而对于当前主流的 PLD 和 FPGA 器件来说，这样的性能很容易达到。

本章参考文献［2］是介绍关于门电路延迟限度的论文，有兴趣的读者可以自己学习。

9.13　用微处理器系统产生等待周期

一些微处理器系统可以通过中央处理芯片产生“等待周期”，并将其插入到存储读写循环中去。

图 9.25 是关于一个输入或者输出（I/O）时序循环的例子（这里进行了必要的简化，但是突出了其中的流程化的部分）。在其中，假设每一个存储单元，或者说 I/O 的循环由 4 个周期（T）组成，它们通过系统时钟 c 产生。T1 周期内是地址设定，T2 周期准备读或者写，T3 周期负责等待数据总线准备完毕，T4 周期用来读或者写数据。图中的事件 FSM 控制器负责监视片选信号 ce，当某一个 I/O 器件被微处理器软件选中时，ce 信号将被拉低。这个过程将在 T1 周期内完成。在 4 个周期中的 T2 周期，输入/输出的写信号 w 或者读信号 r 都会被微处理器拉低。

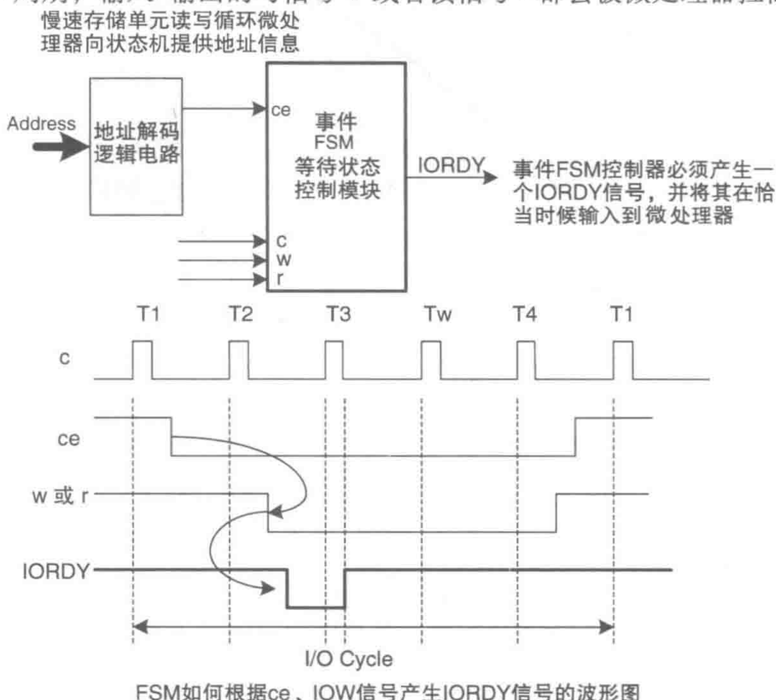


图 9.25　系统框图和 IO 读写循环时序图

在 T2 周期里，FSM 会向微处理器输出一个特殊信号（IODRY），如果微处理器在 T2 周期内检测到这个信号被拉低，那么它将在 T3 和 T4 两个周期之间插入一个新的周期。

这个额外的周期被称为等待周期（Tw），它有效延长了在 T3 周期内一些性能较低的器件准备数据的时间，这样当 T4 到来时，数据的读写才能顺利进行。通过这种方法，状态机控制器就能检测到低性能器件的 ce 信号，并产生一个等待周期。为了确保上述过程的执行，在使用微处理器之前必须仔细阅读器件手册，并了解如何产生等待周期。

事件 FSM 控制器存在的目的就是确定何时将 IORDY 信号拉低，然后何时再拉高。事实上，FSM 要做的事情就是根据图 9.25 中的时序图，找到激活 IORDY（低有效）的时间点，然后送往微处理器。

以时序图作为参考，可以获得状态图（见图 9.26）。从图中可以看出，FSM 的工作顺序是在状态 s0 检测 ce 信号被拉低，读信号 r 或者写信号 w 被拉低，条件都满足后在周期 T2 将 IORDY（低有效）信号拉低。随后 FSM 在状态 s1 检测时钟信号 c 的下降沿，在状态 s2 检测时钟信号的上升沿（由此确定进入周期 T3 的时刻）。随后 FSM 必须检测到时钟信号 c 再次被拉低，指示 IORDY 被拉高的时间点。

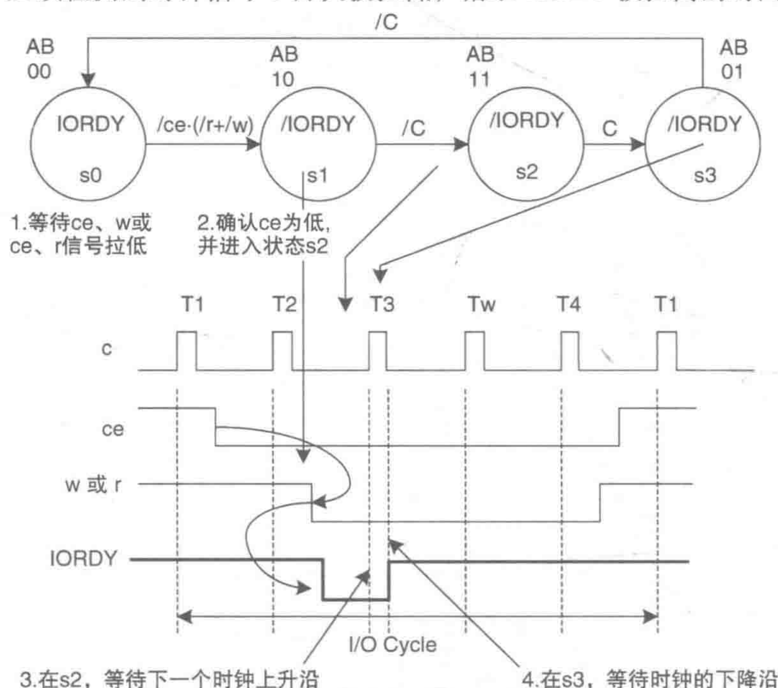


图 9.26 如何从时序图推导系统状态图

注意到快速内存读写循环不需要激活等待周期，因为这些存储芯片的使能信不会和事件 FSM 控制器进行相连。

图 9.27 给出系统的时序公式和输出公式。这个示例向我们展示了如何使用

件 FSM 来跟踪时序信号的变化，同时它的设计方案可以很方便地嵌入到微处理器中去。不过在这之前必须先仔细阅读器件手册，因为不同的微处理器对于低性能存储器件都有各自不同的解决方案。

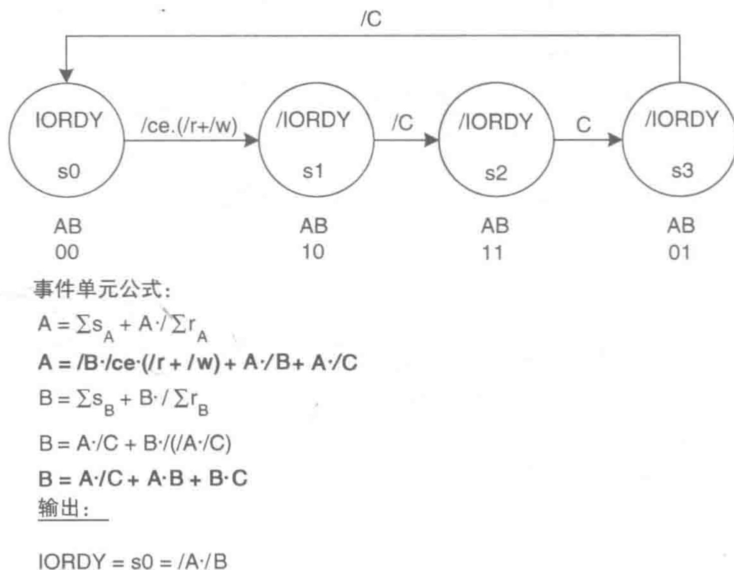


图 9.27　系统时序公式

9.14　用异步 FSM 设计甩干系统

系统框如图 9.28 所示。转动电机会带动缸体进行高速旋转，将衣服里多余的水分通过离心力的作用甩出。衣服里的水分通过水泵排出。系统自带一个水位传感器，用来在开始甩干前检测水位是否过高，避免出现电机过载现象。

用户将湿衣服放进甩干机，然后按开始按钮 st。伴随着开始按钮的释放，水泵开始工作。当水位低于传感器的检测位置时，甩干电机开始工作，并带动一个计时器（这里图中没有显示出来）。

经过一段时间之后，计时器计时完毕，系统让甩干电机和水泵都停止工作。随后一个指示灯将被点亮，告诉用户甩干结束。如果还需要再次进行甩干，用户此时必须按下停止按钮 sp，然后才能重复上述过程。这个系统没办法检测甩干机缸体的门是否在甩干之前被关闭。读者可以自己试着将这个功能加入系统。

图 9.29 是状态图。按下开始按钮后，系统开始测试缸体内水位是高于还是低于水位传感器。如果高于传感器，FSM 经过状态 s1 进入状态 s2，打开水泵。

注意水泵只有在开始按钮被释放之后才能工作。一旦水位降到传感器检测位置之下，FSM 进入 s3，甩干电机开始工作，同时启动计时器。计时结束后，FSM 进入状态 s4，将甩干电机和水泵关闭，点亮指示灯 D。注意 FSM 一旦进入 s4 后就被

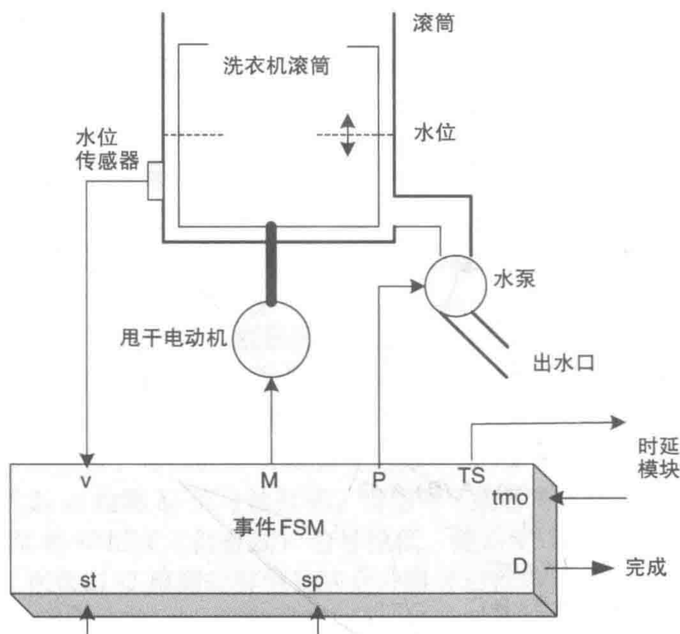


图9.28　甩干机和FSM组成的系统框图

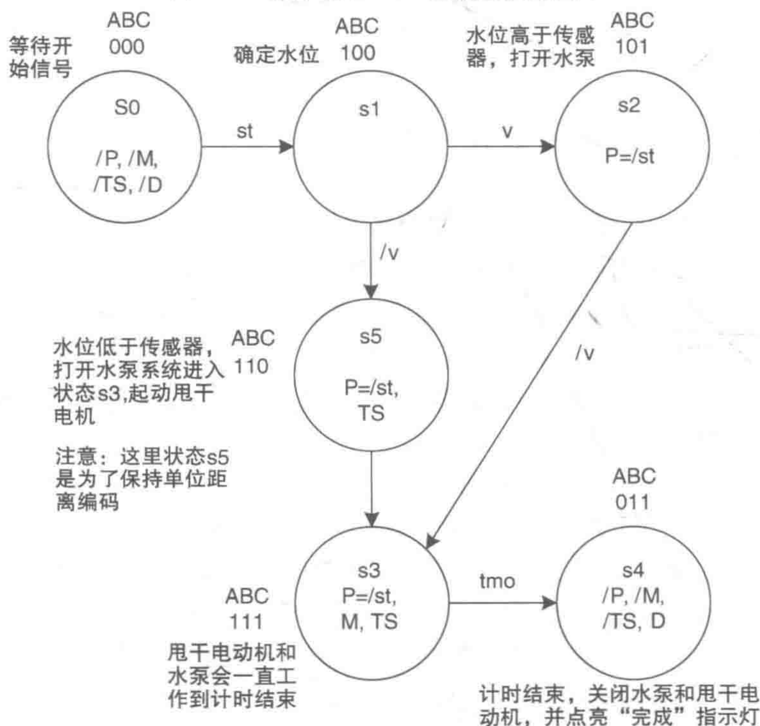


图9.29　甩干机系统的状态图

锁止。事实上，停止按钮可以作为复位输入信号来用，并可以随时让系统停工作。

如果在一开始就检测到水位是低于传感器检测位的，FSM 的状态切换路线是 s0 到 s1 再到 s5，然后进入 s3。状态 s5 存在的意义是确保单位距离编码的运用，而它事实上是一个冗余状态。

从状态 s5 到状态 s3 之间是没有输入条件的。这表明当 FSM 进入 s5 之后会立刻直接进入 s3，两者之间的延迟取决于系统处理两个事件模块的传输延迟，先是 B 然后是 C。

事件模块的公式为：

$$
\begin{aligned}
A &= \sum s_A + A \cdot \sum / r_A \\
&= /B \cdot /C \cdot st + A \cdot /(B \cdot C \cdot tmo) \\
&= /B \cdot /C \cdot st + A \cdot /B + A \cdot /C + A \cdot /tmo \\
B &= \sum s_B + B \cdot / \sum r_B \\
&= A \cdot /C \cdot /v + A \cdot C \cdot /v + B \\
&= A \cdot /v + B \\
C &= \sum s_C + C \cdot / \sum r_C \\
&= A \cdot /B \cdot v + A \cdot B + C \\
&= A \cdot v + A \cdot B + C
\end{aligned}
$$

输出信号的公式为：

$$
\begin{aligned}
P &= s2 \cdot /st + s3 \cdot /st + s5 \cdot /st \\
&= A \cdot C \cdot /st + A \cdot B \cdot /st
\end{aligned}
$$

$$M = s3 = A \cdot B \cdot C$$

$$TS = s5 + s3 = A \cdot B$$

$$D = s4 = /A \cdot B \cdot C$$

输入信号 sp 和 A、B 和 C 分别相与，这样 FSM 可以在 sp 变为逻辑 0 的时候回到初始态 ABC = 000。

代码 9.5 给出了整个系统的 Verilog 语言对应的源代码，其中将公式注释掉了，用门级关系替代了它们。

```
///////////////////////////////////////////////////////////////
          //////异步 FSM 控制甩干机和水泵///////
///////////////////////////////////////////////////////////////
module smpfsm(st,sp,v,tmo,P,M,TS,D,A,B,C);

input st,sp,v,tmo;
output P,M,TS,D,A,B,C;
wire w1,w2,w3,w4,w5,w6,w7,w8,w9;
//图 9.31 对应的公式
```

```
//assign
 //A = (~B&~C&st | A&~B | A&~C | A&~tmo)&sp,
 //B = (A&~v | B)&sp,
 //C = (A&v | A&B | C)&sp,

   //对应图 9.32 的门级描述
   //每个逻辑门被赋予 5 个时间单位的延迟

   or #5 g1(A,w1,w2,w3,w4)'
   and #5 g2(w1,~B,~C,st,sp);
   and #5 g3(w2,~B,A,sp);
   and #5 g4(w3,~C,A,sp);
   and #5 g5(w4,~tmo,A,sp);
     //-------------------------
   or #5 g6(B,w5,w8);
   and #5 g7(w5,A,~v,sp);
   and #5 g11(w8,B,sp);
     //--------------------------
   or #5 g8(C,w6,w7,w9);
   and #5 g9(w6,A,v,sp);
   and #5 g10(w7,A,B,sp);
   and #5 g12(w9,C,sp);

P = A&C&~st | A&B&~st,
M = A&B&C,
TS = A&B,
D = ~A&B&C;

endmodule
/////////////////////////////////////////////////////////////////
```

代码 9.5 甩干机对应的 Verilog FSM 源代码

测试模块在代码 9.6 里。

```
timescale 1ns/10ps
module test;
   reg st,sp,tmo,v;
   smpfsmuut(st,sp,v,tmo,P,M,TS,D,A,B,C);
initial
   begin
```

```
  sp = 0;
  st = 0;
  v = 0;
  tmo = 0;
   ///////
   #10 sp = 1;//去掉复位状态
   #10
   #10 v = 1;//缸体中水位较高
   #10
   #10 st = 1;//系统开始工作
   #10 //这里 FSM 应该进入 s1,然后进入 s2
   #10 st = 0;
   #10 //打开水泵排水
   #10 //等待水被抽干
   #10 v = 0;//指示水已经被抽干
   #10 //应该进入状态 s3,甩干电机开始工作
   #10
   #10 //等待计时结束并关闭甩干电机
   #10 tmo = 1;//关闭甩干电机信号
   #10 //应该进入状态 s4
   #10 tmo = 0;
   #20 st = 0;//将开始按钮释放
   #10 sp = 0;//系统停止工作,状态机复位
   #20
   #20sp = 1;//复位信号释放
   #10 st = 1;//系统再次开始工作,缸体内没有水
   #20
   #20 st = 0;
   #20 //FSM 进入 s1,然后 s5,然后 s3
   #10 tmo = 1;//计时结束,应该进入 s3
   #20 //等待用户将系统关闭
  #10 $stop
  end
endmodule
```

代码 9.6　Verilog 测试固件模块

图 9.30 是用时序公式描述而产生的仿真结果。其中，某些时候 3 个事件模块 A、B 和 C 几乎是在同一时刻改变状态，但事实上，这其中的变化还是按照一定顺

序来的，只是没办法分辨而已。不过，这里必须考虑到传输延迟满足 33.3% 的规则，以防止 9.12.3 节中讨论的竞争冒险的出现。

图 9.31 是用门级描述而产生的仿真结果。由于每一个逻辑门都带有 5 个单位时间的延迟，所以状态切换的过程看得比较清楚。例如 s1（ABC = 100）和 s2（ABC = 101）之间的转换，还有 s1（ABC = 100）进入 s5（ABC = 110），然后进入 s3（ABC = 111）。图中虚线帮助大家看清楚每个状态切换过程中二次状态变量变化的先后顺序。

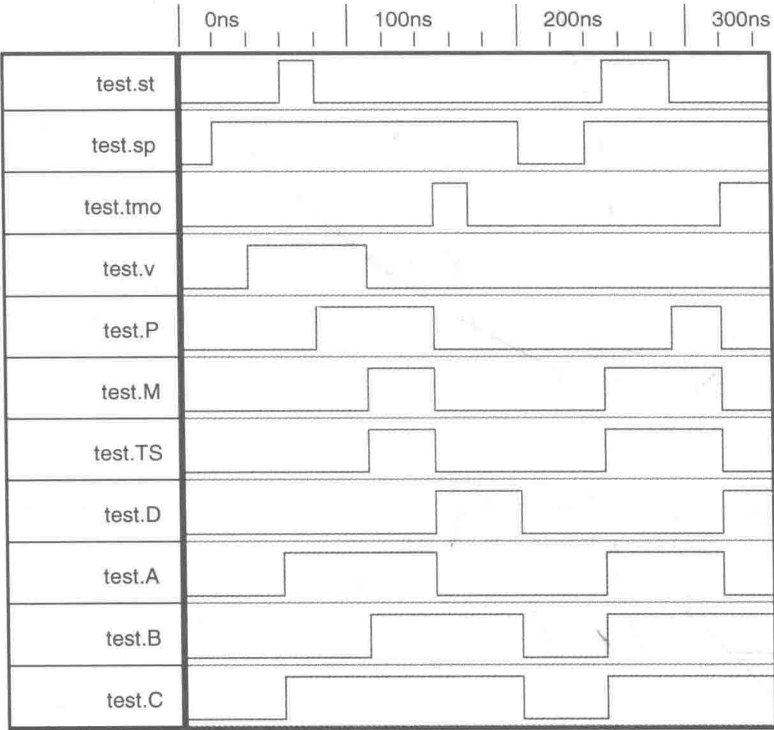


图 9.30 用时序公式描述甩干机的仿真图

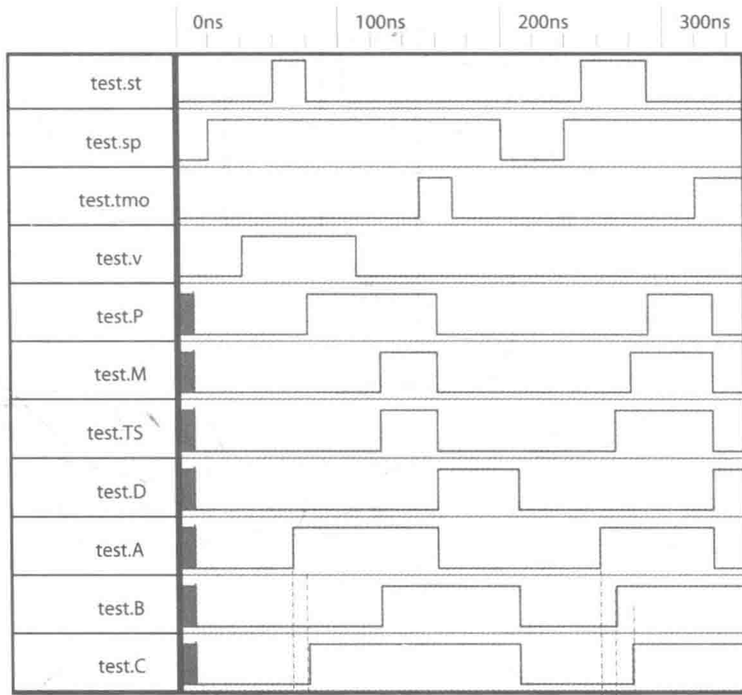


图9.31 用门级描述甩干机的仿真图

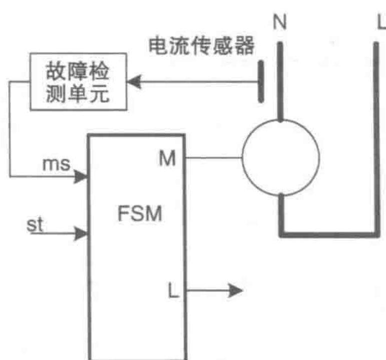
9.15 使用两路分支要注意的问题

图9.10所示的状态图中，在状态s1出现了两路分支的情况，一路以/st为条件返回状态s0，另一路以ms+t为条件进入状态s2。这两个分支所对应的输入信号条件必须相互独立，没有任何重合，否则FSM会出错。如果这种情况无法避免，FSM就需要重新设计，状态的切换就只能针对一个输入条件来完成。

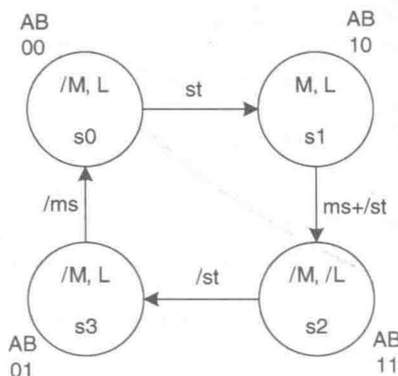
图9.32是电机控制器系统的另一种设计方案（没有测试输入信号t）。新的方案里，FSM从s1到s2的条件有两种，一个是输入信号st回到逻辑0，或者另一个输入信号ms变为逻辑1。如果因为出现故障而进入状态s2，电机会停止工作，故障指示灯L会被点亮（低有效）。如果输入信号st此时回到逻辑0，指示灯此时会被熄灭，但是FSM如果要回到状态s0还必须等输入信号ms回到逻辑0。

A和B的公式分别为：

$$
\begin{aligned}
A &= \sum s_A + A \cdot \sum / r_A \\
&= s0 \cdot st + A \cdot /(s2 \cdot /st) \\
&= /B \cdot st + A \cdot /(B \cdot /st) \\
&= /B \cdot st + A \cdot /B + A \cdot st
\end{aligned}
$$



a) 系统框图



b) 状态图
改进后的状态图优化了两路分支所可能引起的电路误操作

图 9.32　9.6.2 节电机控制器改良版
a) 系统框图　b) 状态图

$$B = \sum s_B + B \cdot / \sum r_B$$
$$= s1 \cdot (ms + /st) + B \cdot /(s3 \cdot /ms)$$
$$= A \cdot ms + A \cdot /st + A \cdot B + B \cdot ms$$

输出信号的公式和图 9.10 里给出的一样。

现在再看几个两路分支的例子，在 9.10.1 节，图 9.20 里有两种两路分支的能性：一个在状态 s1，另一个在状态 s3。每个分支都对应不同的输入条件，并很容易导致系统错误。不过图 9.21 把这样的问题解决了。

在 9.11 节，图 9.22 里状态 s1 也面临一个两路分支。如果输入 sp 在状态 s1为逻辑 1，FSM 在 st = 0 时可以回到状态 s0，也可以在 st = 1 时进入状态 s2。然输入信号 st 和 sp 在状态 s0 时，如果同时从逻辑 0 变为逻辑 1，那么可能发生的化总结为下面两张表格：

st	sp	
0	0	在状态 s0
0	1	sp 在 st 之前变化；停在状态 s0 等待信号变化
1	1	留在状态 s0

或者

st	sp	
0	0	在状态 s0
1	0	st 在 sp 之前变为逻辑 1；FSM 从 s0 进入 s1
1	1	进入状态 s2

第二张表格的工作状态看起来是正常的。

通常来说，两个或两个以上输入信号同时变化会导致电路不稳定，这是因为输入信号之间传输延迟的不统一，会导致静态或者动态竞争冒险。最好的解决办法就是将状态的切换条件受制于一个输入信号的变化，而不是多于一个。图 9.33 给出了一个解决方案。

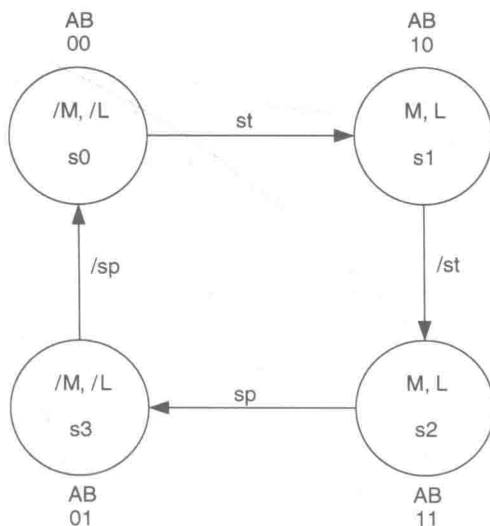


图 9.33　图 9.22 中状态图改良版

新的状态图和原本的设计意图显然是不符的。事实上图 9.22 里的例子里所描述的原始的设计方案很难真正用于实践。

设计一个带有多个输入信号变化的异步 FSM 并不是一件很容易的事情，并且

本书也不会再深入讨论下去。感兴趣的读者可以参考本章最后的参考文献 [3]，里面详细介绍了如何运用霍夫曼（Huffman）和穆勒（Muller）电路来设计复杂异步 FSM。需要特别指出的是，C 型逻辑门的运用可以抵消触发条件和释放条件。这样当两个或两个以上输入信号发生变化时，潜在的静态和动态竞争冒险可以被削弱。

9.16 小结

本章详细介绍了异步（事件触发）FSM 以及如何使用 PLD 和 FPGA 器件来实现。同时简要介绍了带有继电器的电路系统。其中涵盖了如何运用 Verilog HDL，通过公式和基本逻辑门对不同的设计方案进行简单的仿真。设计方案直接通过公式或者逻辑门来实现，避免了大部分硬件系统在行为级设计时，运用事件触发而产生的各种问题。此外还介绍了一些简单的事件 FSM 的运用。对于事件触发可能产生的竞争冒险问题本章也单独做了讨论，并给出了避免竞争冒险的方法。

参考文献

1. Zissos D, Duncan FG. Problems and solutions in logic design, 2nd edn. Oxford University Press, 1979.
2. Duncan FG, Zissos D. Gate tolerance in sequential circuits. Proc. IEE 1971;118(2):317–320.
3. Myers C. Asynchronous circuit design. John Wiley & Sons, Ltd, 2001.

第 10 章　佩特里（Petri）网络

10.1　简易佩特里网络概述

佩特里网络是一种可以用来描述时序和并行系统行为的状态框图。最初它是由卡尔．佩特里（Karl Petri）在 20 世纪 60 年代构思出来的，并得到了一大批学者的追随。这里向大家介绍一个关于佩特里网络的网站，http：//www. informatik. uni-hamburg. de/TGI/PetriNets/，里面关于佩特里网络的内容较为全面。

佩特里网络通常被用作研究并行系统（不一定是电子系统）的一系列行为。对于并行系统编程的研究也借鉴了佩特里网络的理论。近些年来，一些学者开始向世人展示如何运用佩特里网络，像设计、综合同步和异步状态机系统那样设计并综合有限状态机系统[1]。而运用佩特里网络的主要目的是为了设计并行系统。因此本章以下内容主要参考了文献［1］。

图 10.1 是一个只有两个状态的 FSM 框图，以及运用佩特里网络来产生的等同系统。根据佩特里网络的定义，FSM 里的“状态”这里称之为“占位符”，FSM 里不同状态之间的“传输路径”这里称之为“弧线”，它在占位符（P1 和 P2）和传输点（T1 和 T2）之间建立连接。FSM 里传输线上的输入条件在佩特里网络中被放在传输点上，而传输点的位置处在连接两个占位符的弧线的中央。

佩特里网络的每一个占位符都占用系统的一个存储空间（稍微有点类似于独热系统的构架）。不过，在佩特里网络中可以存在多个有效的占位符（而在 FSM 里同一时间只能有一个状态有效）。基于这个特性，我们需要知道在某一个时刻哪些占位符是有效的。方法是用一个“标记”来显示一个有效的占位符，具体做法是在占位符中放置一个“圆点”。

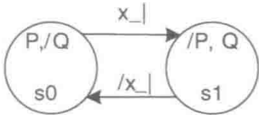
图 10.1 中，占位符 P1 处于有效状态，因为它带有标记；占位符 P2 不带标记，因此它未被激活。

下面简要描述一下佩特里网络的行为模式。

一开始，标记出现在占位符 P1（网络的初始化将在后面介绍）。当输入信号 x 有效（即 $x=1$）时，网络启动传输 T1，标记将（沿着 T1 弧线的方向）移动到占位符 P2，随后停留在 P2 中（因为此时 x 仍然为 1，传输 T2 无法启动），过程如图 10.2 所示。

要注意的是，传输 T1 只有在 $x=1$ 的条件得到满足，并且有一个时钟脉冲到来时才会触发。而且输出信号在占位符 P2 里的状态是 $P=0$，$Q=1$，因此这是一个摩

采用独热编码设计的状态图



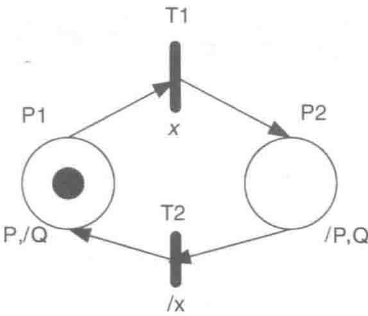
$s0\cdot d = s1\cdot /x + s0\cdot /x$

$s1\cdot d = s0\cdot x + s1\cdot x$

$P = s0$

$Q = s1$

Petri网



$P1\cdot d = T2 + P1\cdot /T1$

$P2\cdot d = T1 + P2\cdot /T2$

$T1 = P1\cdot x\cdot /P2$

$T2 = P2\cdot /x\cdot /P1$

$P = P1$

$Q = P2$

更复杂的逻辑

图 10.1　状态机和佩特里网络的比较

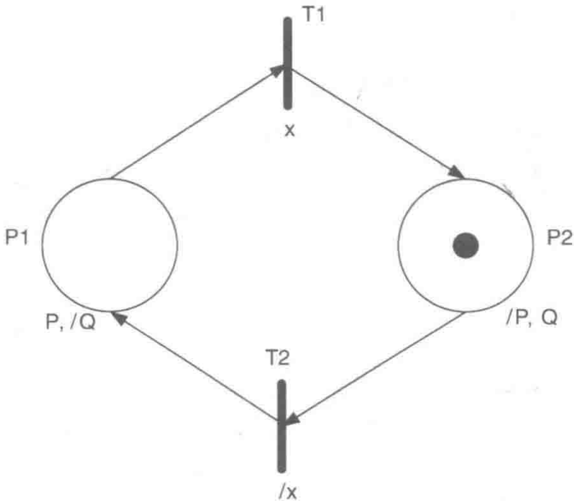


图 10.2　标记在 T1 启动后（x = 1）移动到 P2

尔（Moore）型模块。当 x = 0，且下一个时钟脉冲到来时，标记将回到占位符 P1。

关于网络的综合是基于图 10.1 里的公式来完成的。公式的基本类型有三种：

- 占位符公式；
- 传输公式；
- 输出公式。

占位符公式的格式和事件 FSM 的时序公式一样，如图 10.1 里 P1 · d 和 P2 · d 所示。佩特里网络的公式将输入信号定义成 D 触发器的输入，因此“P · d”出现在等号的左边：

$$P1 \cdot d = T2 + P1 \cdot /T1 \tag{10.1}$$

这里的公式可以理解成：让 P1 得到一个标记，必须启动 T2；又或者，为了持有一个标记，此刻标记必须已经处于 P1，且 T1 不能被启动。

对于占位符 P2，公式为

$$P2 \cdot d = T1 + P2 \cdot /T2 \tag{10.2}$$

式（10.1）右边第一项 T2 对于 P1 来说，是占位符有效的激活条件。乘积项 P1./T1 是占位符的状态保持条件。

传输公式则由能够启动传输的必要条件组成。从图 10.1 中可以看出，佩特里网络的 T1 只有在 P1 拥有了标记，P2 没有标记，且 x = 1 等条件均满足的情况下才会触发。所以公式可以写为

$$T1 = P1 \cdot x \cdot /P2 \tag{10.3}$$

同样地

$$T2 = P2 \cdot /x \cdot /P1 \tag{10.4}$$

对于更加复杂的佩特里网络，公式成立所需要的条件也就更加繁多，将在后续讨论。

图 10.3 里电路图上 P1 · d 和 P2 · d 分别对应两个 D 触发器的输入端。

这里需要强调的是，后续的例子中占位符的公式将以递归的形式出现，而不是 Pn · d 的形式，例如：

$$P1 = T2 + P1 \cdot /T1$$

$$P2 = T1 + P2 \cdot /T2$$

这表明公式左边是触发器的输入信号。读者可以参阅参考文献［1］。

图 10.3 给出了完整的佩特里网络的框图设计、公式以及最终的综合电路图。因此一旦佩特里网络设计完成，系统综合只是一种规则的应用而已。

显然，可以直接运用 PLD 或者 FPGA 器件以及公式来进行设计，或者用 Verilog HDL 来描述其行为。

注意图 10.3 里电路图的初始化设定，这和之前的独热编码 FSM 是一样的。同时大家还要留意逻辑门的设计构架。触发器输出 P1 作为反馈，成为与门的其中一个输入；同样 P2 也作为反馈输入到另一个与门。两者的作用是相同的，使得占位符保持有效状态。

从上述的电路图和公式描述，可以总结出触发器在其中的作用是存储占位符的

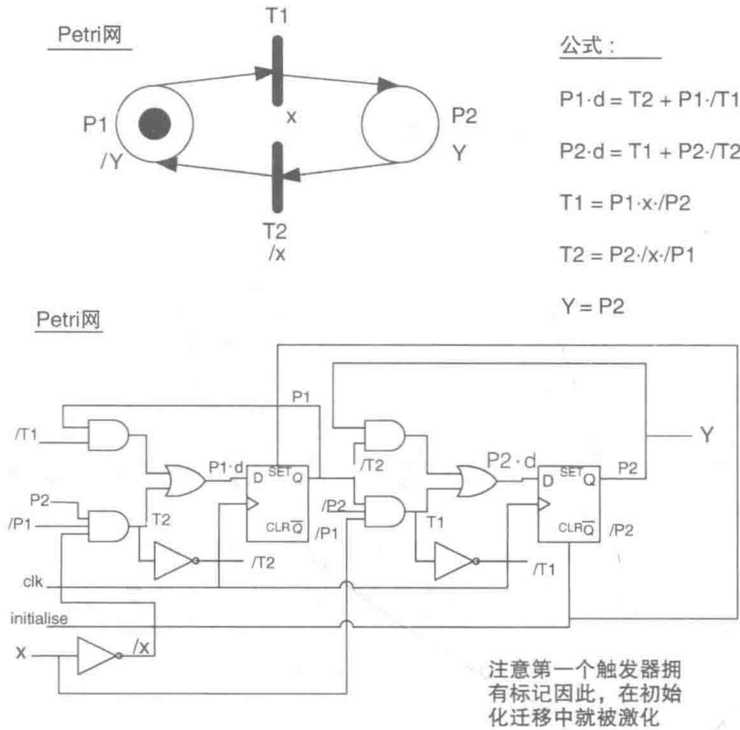


图 10.3　佩特里网络到电路图的完整过程

状态。带有使能的触发器类似于带有标记的占位符，而带有复位的触发器类似于不带标记的占位符。

图 10.4 给出了佩特里网络的基本局部构造：

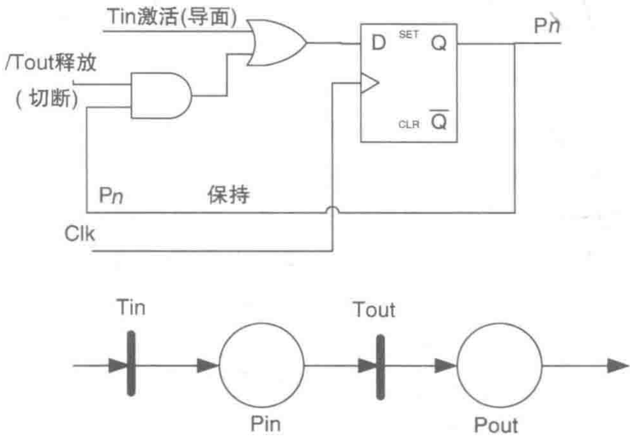


图 10.4　佩特里网络基本局部构造

$$Pn = Tin + Pn \cdot /Tout \tag{10.5}$$

Tin 是激活输入，输出信号 P*n* 反馈到输入端，作为与门的其中一路输入信号，形成状态保持项。式（10.5）中的 Tin 的格式为：Tin = 输入占位符 & 输入使能 &/输出占位符

Tout 是释放项，在上述公式里被取反。当 Tout 被拉高时，/Tout 的值将变为逻辑 0，以便打破反馈信号形成的状态保持，将 D 触发器复位（Tin 此时将不在被激活状态）。

如果将 D 触发器从图 10.4 里移除，图中与门和或门以及反馈信号加在一起，正好形成一个异步事件模块，如图 10.5 所示。因此佩特里网络可以是同步的，也可以是异步（事件触发）的。

注意如果要用异步系统来设计，逻辑门之间的延迟必须纳入考虑，这和第 9 章讨论的异步 FSM 所涉及的内容相似。

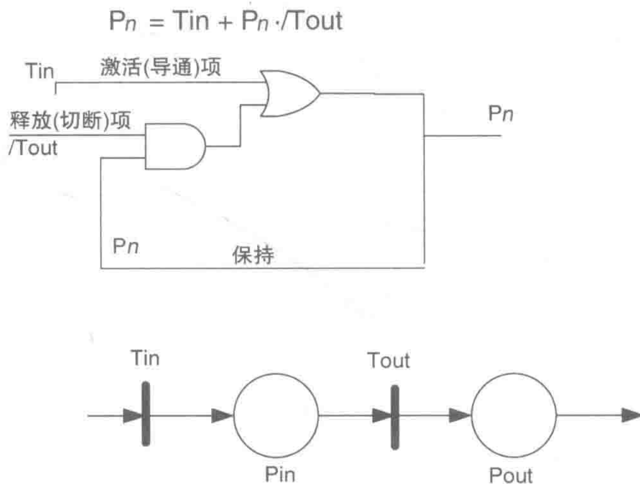


图 10.5　异步（事件触发）佩特里网络结构

对于佩特里网络：

- 同步设计是用时钟驱动，配合 D 触发器来完成；
- 异步设计是用事件触发，不含有 D 触发器。

对于异步佩特里网络有一些大学正在从事相关的研究。读者可以通过网络搜索关键字“佩特里网络”和“C 逻辑门”来获得更多的信息。

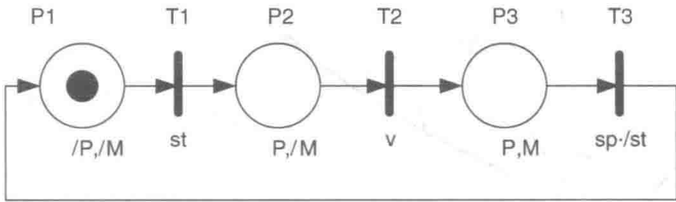
本章余下的部分将重点讲解时钟驱动的同步系统。

为了巩固前面所讲的内容，来看一个关于佩特里网络时序控制器的例子。

10.2　使用佩特里网络设计简单时序逻辑

图 10.6 是一个关于佩特里网络时序控制器的例子。其中，水泵 P 可以通过激活信号 st（逻辑 1）并启动 T1 来开启。在传感器 v 变为逻辑 1 之后，T2 将启动并开启电机。按下停止按钮 sp 将启动 T3，系统回到占位符 P1，此时水泵和电机均处于停止状态。

用时序佩特里网络解决水泵—甩干电动机问题



写出控制系统的佩特里网络公式

T1 =
T2=
T3 =

P1 =
P2 =
P3 =

图 10.6　时序佩特里网络

读者可以在图中空白处试着将公式补充完整。图 10.7 中给出了公式的答案并带有系统的电路图。初始化电路也包含在其中，状态为 P1 对应的触发器被激活，P2 和 P3 对应的触发器被复位。

如果让系统变为异步系统，可以将图中的 D 触发器移除，或门的输出作为反馈，成为两个与门的其中一路输入，这样便组成了 P1、P2 和 P3 的事件模块。

设计公式：

T1 = P1·st·/P2　　　　P1 = T3 + P1·/T1
T2 = P2·v·/P3　　　　P2 = T1 + P2·/T2
T3 = P3·sp·/st·/P1·　　P3 = T2 + P3·/T3

P = P2 + P3

M = P3

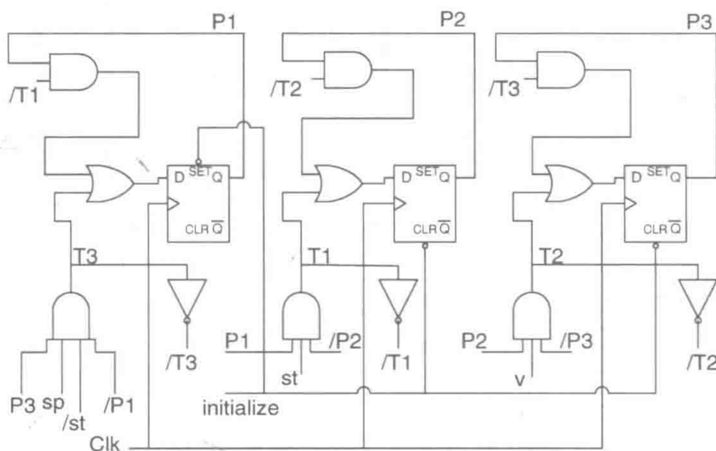


图 10.7　佩特里网络公式和电路图

10.3　并行佩特里网络

讲到这里，只接触到了一个时序佩特里网络。然而，运用佩特里网络的主要目的是为了设计并行系统。现在来讨论用佩特里网络设计并行系统的方法。

并行佩特里网络是含有并行网络的系统。图 10.8 就是这样一个例子。其中在传输点 T2 和 T5 之间含有 3 个并行网络。P1 和 P2 组成一个串行时序网络。在传输点 T2，它们“分化”成三路并行网络。在传输点 T5，三路并行网络“合并”在一起，再次以串行的方式在系统中运行。

当标记到达占位符 P2，且输入信号 syn1 被激活（逻辑 1），标记将分别进入占位符 P3、P4 和 P5，具体如图 10.9 所示。此时系统将同时拥有 3 个事件模块（含有 D 触发器）被触发。

假设此时输入信号 p 被激活（拉高），但是输入信号 q 还没有被激活。系统运行的结果如图 10.10 所示。如果此时输入信号 syn2 被激活（拉高），传输 T5 将不会启动，因为标记还没出现在占位符 P7。

佩特里网络传输启动的条件是占位符必须带有标记，这里 P6、P4 和 P7 都必

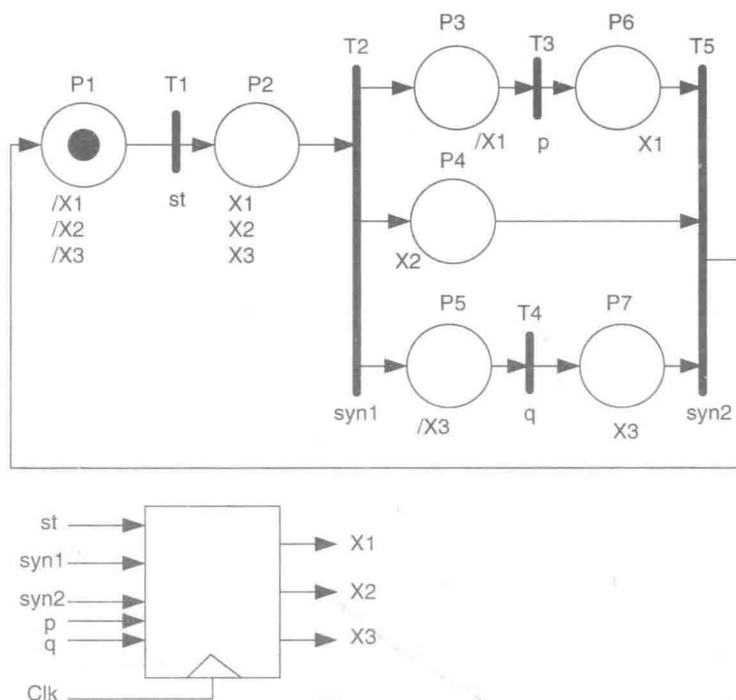


图 10.8　带有并行网络的佩特里网络

须带有标记才能启动 T5。

最终，当输入信号 q = 1，T4 将启动，P5 上的标记将移动到 P7。

在图 10.11 中，所有启动 T5 的占位符均带有标记；所以，一旦 syn2 = 1 条件成立，T5 会立刻启动，标记会融合在一起，标记将再次回到占位符 P1。

上面描述了一种串行网络转为并行，再回到串行的构架。大部分并行系统是这种运行方式，因此佩特里网络可以用来描述类似的行为。而且这也是过去佩特里网络的主要运用场合之一。

从图 10.8 ~ 图 10.11 中，可以看出传输 T2 和 T5 是系统的同步节点；而信号 syn1（控制 T2 的启动）是用来同步“分化”，信号 syn2（控制 T5 的启动）是用来同步“合并”。因此在硬件系统中，信号 syn1 和 syn2 被看作是同步节点。

然而佩特里网络是可以自我调节的，因为所有占位符汇聚到同一个传输之前，必须满足带有标记这个条件。

现在将公式推导出来。

首先是占位符的：

$$P1 = T5 + P1 \cdot /T1$$

$$P2 = T1 + P2 \cdot /T2$$

$$P3 = T2 + P3 \cdot /T3$$

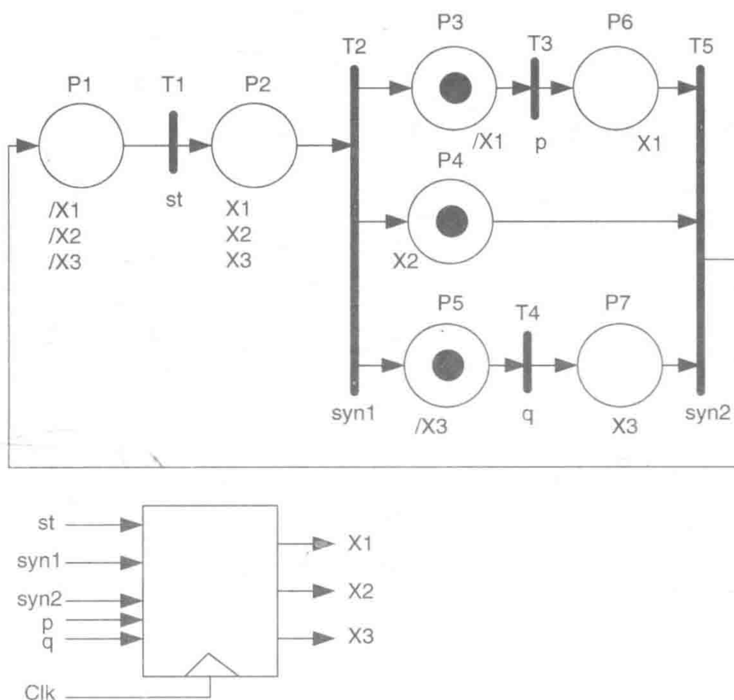


图 10.9　标记移动到三个并行网络（分化）

$$P4 = T2 + P4 \cdot /T5$$

$$P5 = T2 + P5 \cdot /T4$$

$$P6 = T3 + P6 \cdot /T5$$

$$P7 = T4 + P7 \cdot /T5$$

然后是传输点的公式：

$$T1 = P1 \cdot st \cdot /P2$$

$$T2 = P2 \cdot syn1 \cdot /P3 \cdot /P4 \cdot /P5$$

注意到 T2 的启动条件是 P2 必须带有标记，syn1 必须激活，但是 P3、P4 和 P5 都不能带有标记。

$$T3 = P3 \cdot p \cdot /P6$$

$$T4 = P5 \cdot q \cdot /P7$$

$$T5 = P6 \cdot P4 \cdot P7 \cdot syn2 \cdot /P1$$

这里，所有合并到 T5 的占位符必须带有标记。因此对于 T2 和 T5 两个公式在推导时，必须注意把所有条件全部涵盖在其中。

最终，输出信号公式如下：

$$X1 = P2 + P6$$

$$X2 = P2 + P4$$

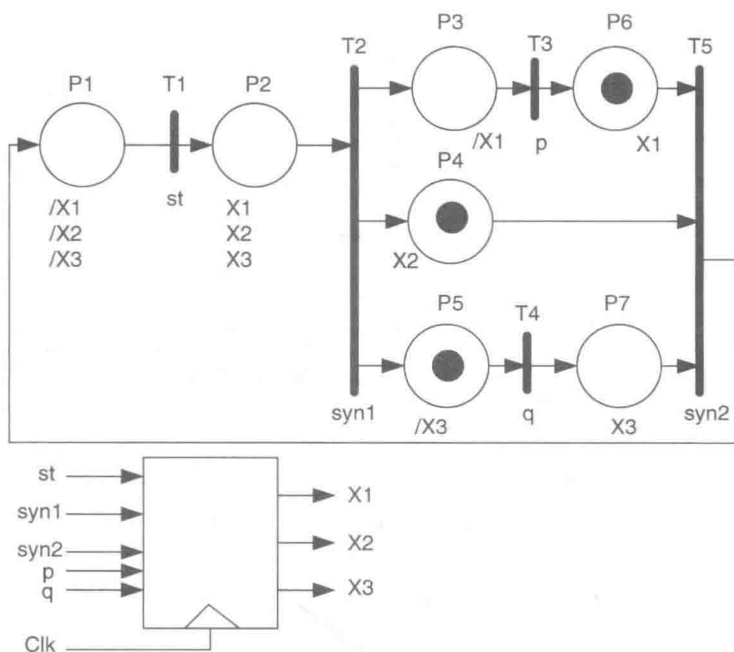


图 10.10　输入信号 p = 1，q = 0，P5、P6 和 P4 带有标记，P7 不带标记

X3 = P2 + P7

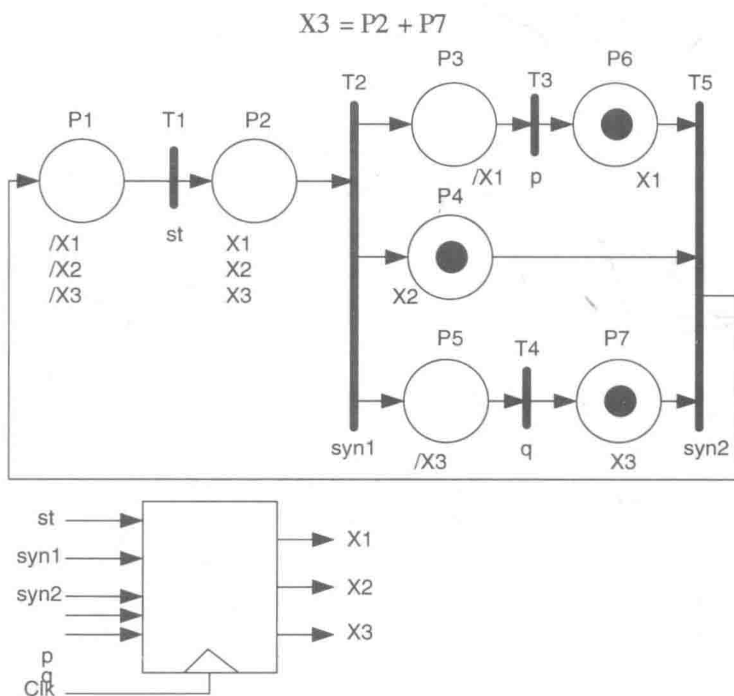


图 10.11　传输 T5 在输入信号 syn2 被激活后就可以启动

10.3.1　另一个并行佩特里网络案例

图 10.12 是另一个佩特里网络的例子。读者可以参照图片写出所有的公式，然后对照下面的答案看是否正确。

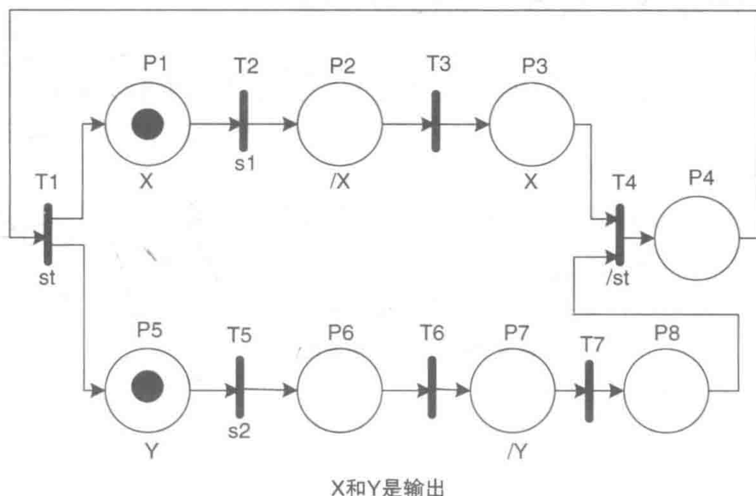


图 10.12　另一个并行佩特里网络

占位符公式：

$$P1 = T1 + P1 \cdot /T2$$

$$P2 = T2 + P2 \cdot /T3$$

$$P3 = T3 + P3 \cdot /T4$$

$$P4 = T4 + P4 \cdot /T1$$

$$P5 = T1 + P5 \cdot /T5$$

$$P6 = T5 + P6 \cdot /T6$$

$$P7 = T6 + P7 \cdot /T7$$

$$P8 = T7 + P8 \cdot /T4$$

传输项的公式为：

$$T1 = P4 \cdot st \cdot /P1 \cdot /P5$$

$$T2 = P1 \cdot s1 \cdot /P2$$

$T3 = P2 \cdot /P3$ 这里没有外来输入信号影响 T3 的启动

$$T4 = P3 \cdot P8 \cdot /st \cdot /P4$$

$$T5 = P5 \cdot s2 \cdot /P6$$

$$T6 = P6 \cdot /P7$$

$$T7 = P7 \cdot /P8$$

输出公式为：

$$X = P1 + P3 + P4$$
$$Y = P5 + P6$$

10.4 并行佩特里网络里的同步传输

10.3 节里，通过同步输入信号 syn1 和 syn2，可以将网络的运行方式从串行转变为并行，然后再回到串行。有时候，需要在两个不同的佩特里网络之间完成同步。图 10.13 就是一个例子。

如果两个网络没有共享的连接关系，同步是很难实现的。这是一个并行程序设计系统中比较典型的问题。不过系统中如果有共用的变量就比较方便了。但仍然存在问题，因为这个共用的变量有可能被两个网络交替赋值。

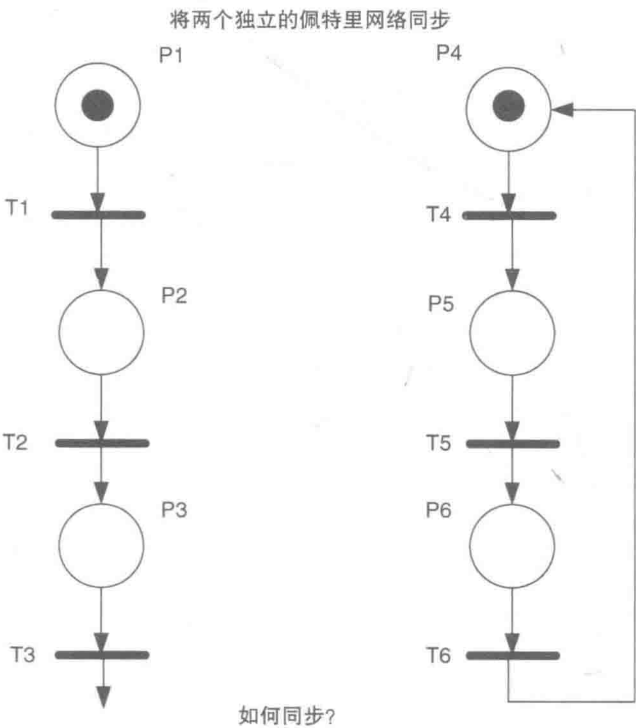


图 10.13 如何同步两个独立的佩特里网络

10.4.1 弧线的有效和失效

在佩特里网络中有一种办法可以解决这个问题，就是使用

- 一个有效的弧线；

- 一个失效的弧线。

先来考虑让一段弧线有效。从图 10.14 里可以看出，从 P1 到 P3 的过程和从 P4 到 P6 的过程是完全相互独立的。不过从 P2 引出的虚线到 T5，表明启动 T5 的条件是 P2 必须带有标记。而且，启动 T5 的条件还必须满足 P5 拥有标记，且信号 go 必须被激活（逻辑 1）。因此，启动 T5 的条件就是：T5 = P2 · P5 · go · /P6 有效弧线的传输公式

这种方案确保了两个佩特里网络在 T5 启动之前，都各自处于一个特定的状态（P2 和 P5）。

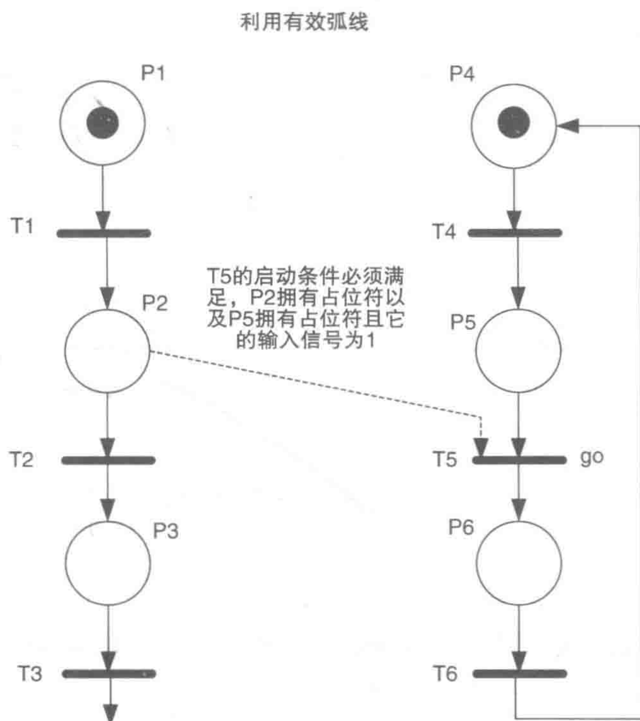


图 10.14 有效弧线

现在来看图 10.15 中让弧线失效的例子。图中佩特里网络 P1 到 P3 分支里，如果占位符 P2 带有标记，此时可以让 P4 到 P6 分支的传输暂停。用公式表达如下：T5 = /P2 · go · P5 · /P6

这里要启动 T5，除了 P5 带有标记，信号 go = 1 之外，P2 不能带有标记。

现在将上述两个概念用一个实例来详细说明。

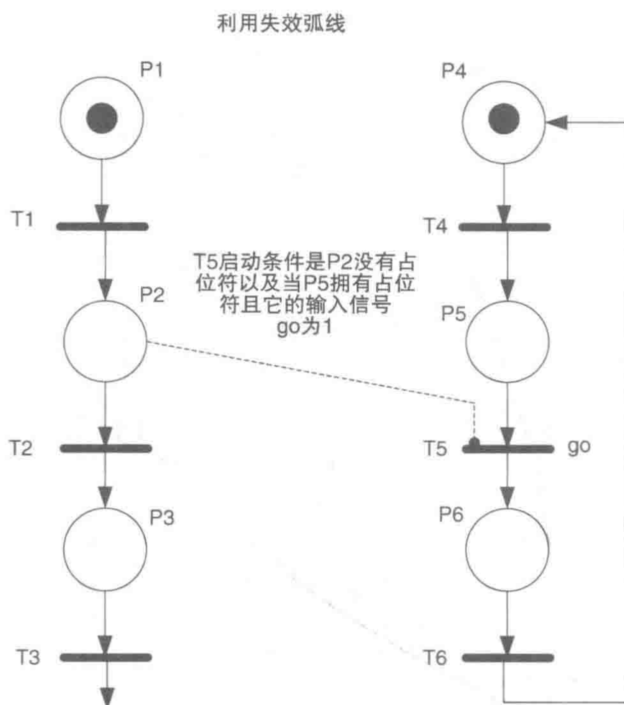


图 10.15　在特定的点让弧线失效避免启动传输

10.5　用有效弧线和失效弧线同步两个佩特里网络

图 10.16 的例子中，系统运行遵循下面一系列表述：

1）系统假设标记总是首先出现在占位符 P5 上，具体原因可能是受到外部因素的影响。

2）在 P2 获得标记之前，P5 里的标记是无法移动到 P6 上的。

3）通过让 P5 的弧线失效，从 P2 到 P3 的转换没法进行，因为 T2 此时无法启动。

4）只要输入信号 go = 1，标记就可以从 P5 移动到 P6。

5）这样就将启动 T2 的限制条件移除了，随后标记就可以从 P2 移动到 P3。

本例说明了弧线的有效和失效对网络传输的影响。

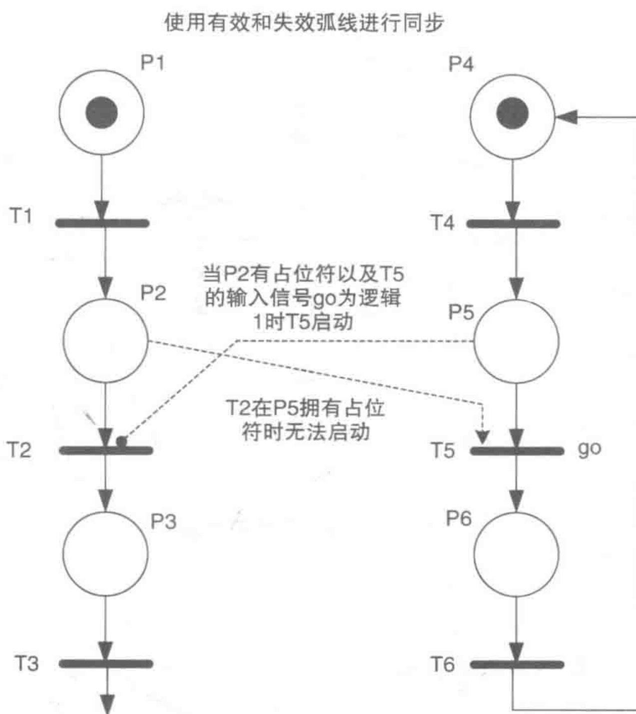


图 10.16　两个独立佩特里网络的优先级配置

10.6　共享资源的控制

现在可以看一个比较实用的例子，如图 10.17 所示，两台计算机 A 和 B 所组成的系统，它们通过同一根总线共享一个资源（例如一台打印机）。两台计算机是通过三态缓冲器来和共享资源隔离，缓冲器是由佩特里网络所产生的 EA、EB 两组信号来控制。输入到佩特里网络的信号为 ra 和 rb，它们分别由两台计算机产生。系统设定计算机 A 的优先级高于计算机 B。

解决这个问题的办法有几种，但是最简洁明了的是图 10.18 中所示的网络构架。图中用了两个独立的佩特里网络：一个处理计算机 A 产生的信号 ra，另一个处理计算机 B 产生的 rb。

如果计算机 A 在计算机 B 产生 rb 信号之前产生信号 ra，P1 上的标记将移动到 P2，并且连接 T3 的弧线在此之后会失效，并将信号 rb 的传输阻断。

随后，计算机 A 将把信号 ra 拉低，标记会回到 P1。如果在计算机 A 访问共享资源期间信号 rb 也被产生，那么 P3 上的标记将不会移动到 P4，因为传输 T3 此时是失效的。

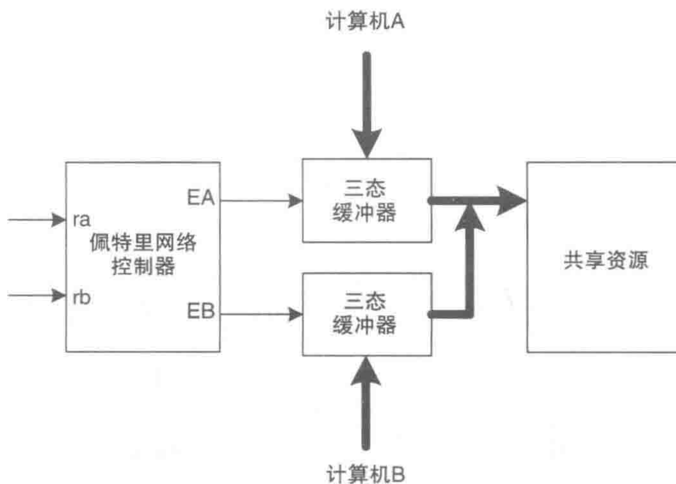


图 10.17 共享资源控制器

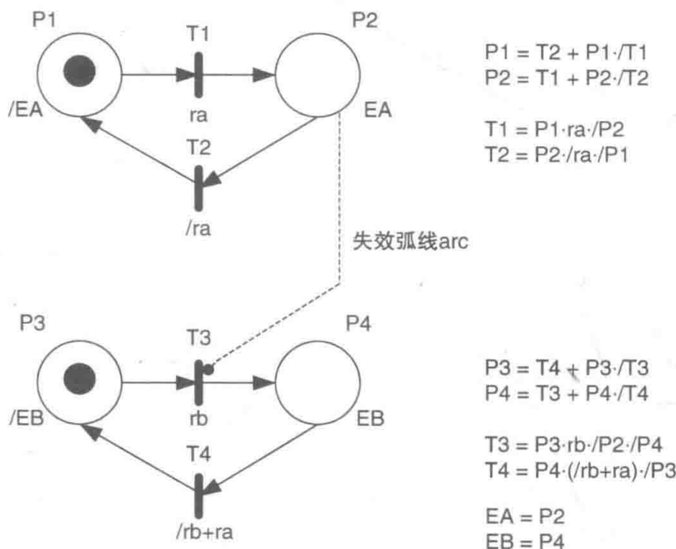


图 10.18 共享资源解决方案

注意当计算机 B 在访问共享资源时，计算机 A 也需要访问，信号 ra 将被拉高并导致 P4 里的标记返回 P3，P1 里的标记移动到 P2。所以这也说明计算机 A 的优先级高于计算机 B。

当然如果计算机 B 在访问共享资源时，计算机 A 没有任何动作，那么当计算机 B 完成访问后，拉低信号 rb，也会让标记从 P4 返回到 P3。

图 10.18 里含有佩特里网络对应的公式。在这里特别提醒大家注意 T3 公式里

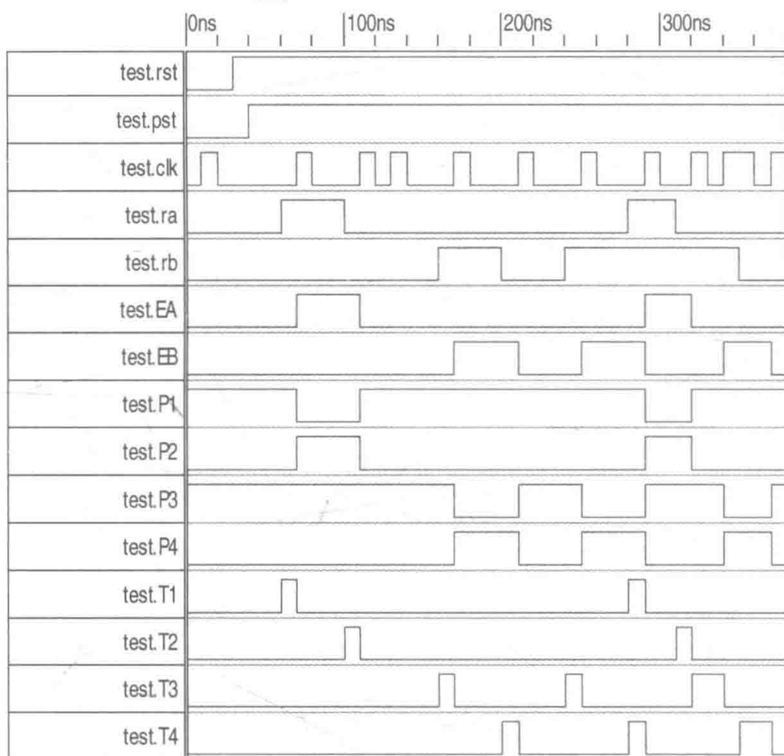


图 10.19　共享资源佩特里网络仿真结果

的/P2 项。T3 被启动的条件必须满足 P3 里带有标记，rb 信号被拉高并且 P4 或者 P2 里没有标记。

在仿真的一开始，由于初始化信号 rst 和输入信号 pst 的作用，占位符 P1 和 P3 是被激活的。输入信号 ra 和 rb 先后被激活并模拟访问共享资源的请求。在第七个时钟脉冲到来时，输入信号 rb 被激活；随后信号 ra 在第八个时钟脉冲被激活（计算机 A 的请求有更高的优先级）。这导致计算机 A 将取代计算机 B 获得共享资源的访问权限。在计算机 A 完成了访问之后，由于信号 rb 仍然有效，计算机 B 重新获得访问共享资源的权限。随后 rb 被拉低，佩特里网络将标记从 P4 返还到 P3，并断开共享资源和计算机 B 之间的连接。

10.7　二进制数据的串行接收器

在 4.7 节，使用 FSM 和 D 触发器设计了一个异步二进制数据接收器，并带有移位寄存器、4 位计数器以及数据锁存器等模块。

本节设计一个佩特里网络控制器再次实现同样的功能。这里将详细介绍整个设

计过程，如果没有读过第 4 章的内容，也不会影响理解。

异步串行接收器在这里的作用是接收二进制串行数据，并将其转换为并行数据。佩特里网络是一种比较好的选择，因为我们可以利用有效弧线和网络构架来进行系统设计。

图 10.20 再次将第 4 章里所描述的数据包，以及传输协议呈现在这里，其中还包含了采样点。异步串行协议含有 1 位起始位（低有效）、8 位数据位、2 位结束位（一共 11 位数据）。输入数据需要进入移位寄存器，并且确保进入移位寄存器的数据是有效的，这是模块设计的关键。达到这个目的的办法，就是需要使用一个比移位寄存器更快的时钟来进行采样，这样确保时钟的脉冲在数据进入移位寄存器时对准数据的中央。

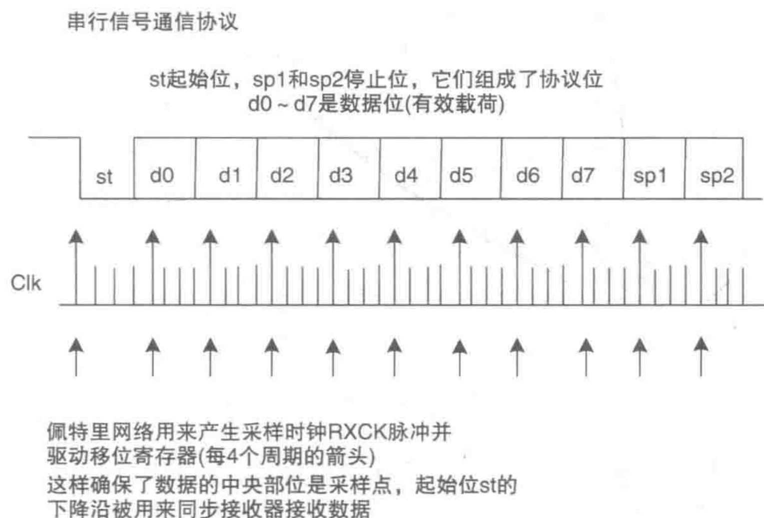


图 10.20 数据包和传输协议

图 10.20 中，采样时钟频率大约是移位寄存器数据速率的 4 倍，即每一位数据的周期对应采样时钟 4 个周期，大约在采样时钟的第二个周期到来时，移位寄存器输入端会将数据加载到寄存器里（如图中的箭头所示）。换句话说，移位寄存器的时钟速率是 FSM 速率的 1/4。这里用佩特里网络的内部时钟产生移位寄存器的时钟。

图 10.21 是基于佩特里网络的系统框图。图中佩特里网络控制器代替原来的 FSM，控制整个系统的操作，其中包括并行输出的 11 位移位寄存器和数据锁存器。注意连接数据锁存器的只有 8 位数据（Q0 ~ Q7），并不包括起始位和停止位。4 位计数器（可以是异步二进制计数器，也可以是同步二进制计数器）用来统计加载的数据位数并输出信号 rxf，通知佩特里网络移位寄存器已经存满。移位寄存器的时钟 RXCK 信号，是由佩特里网络的内部时钟产生的。

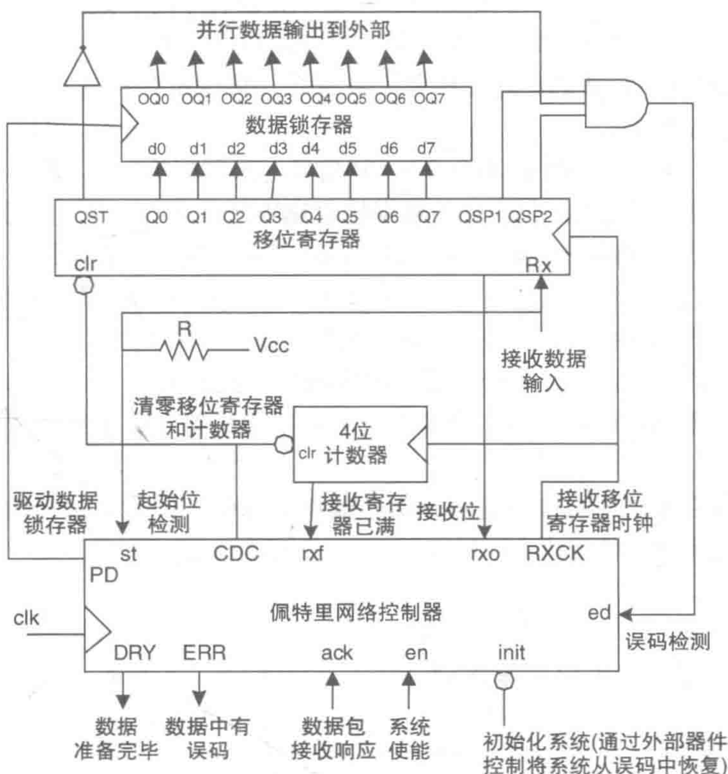


图 10.21 异步串行接收系统框图

一旦数据接收出现错误，信号 ed 将提醒佩特里网络，同时佩特里网络也将输出信号 ERR 拉高，此时系统将等待外部器件重新初始化（复位）控制器，并准备再次接收数据。这里的外部控制也会通过串行接收系统来完成。整个系统和第 4 章所介绍的非常相似。

系统在开始信号被拉低时启动。图 10.22 是佩特里网络对应的系统框图。它由两个独立的佩特里网络组成，之间用有效弧线连接。第一个网络，含有占位符 P1 ~ P5，用来产生移位寄存器的时钟 RXCK。第二个是主网络，用来控制系统的操作流程。两个网络都由系统时钟 clk 来驱动。

留意网络中的 3 条有效弧线。第一条弧线，从 PM2 开始，作用是当主网络收到开始信号 st 时，传输 T1 才能启动。第二条弧线，从 P5 开始，作用是在第一个网络产生移位寄存器时钟脉冲 RXCK 之前，不能让主网络进入 PM3。还有一条控制弧线是用来控制 T5 的启动，只有当主网络移动到 PM3 之后，T5 才具备启动条件。否则，在 T5 和 TM2 之间会存在潜在的竞争冒险。

基于这种机制，第一个佩特里网络，可以产生跟数据包同步的移位寄存器时钟。注意佩特里网络产生的寄存器时钟周期相当于 5 个系统时钟周期，而不是图

10.20 里的 4 个。因此系统时钟的采样频率必须是波特率的 5 倍。

在主网络中，占位符 PM3 以及它所延伸出去的传输 TM3 和 TM7，被用作测试移位寄存器满格信号 rxf 的状态。如果信号为低（移位寄存器没有存满），那么主网络会返回占位符 PM2。

注意当 rxf = 0 时，PM5 将不会产生 PD 信号（米利型输出）。而且 TM5 在 rxf = 0 时启动。一段时间后当一包完整的数据接收完毕之后（11bit），主网络将移动到 PM4，检测 ed 信号的状态。如果 st、sp1 和 sp2 信号的接收是正常的，ed 信号此时应该是逻辑 1。此后主网络会进入占位符 PM5，如果此时 rxf = 1，系统会产生 PD 信号来锁定接收到的数据，并将它们送到锁存器以便外部取用。

然后主网络将在此等待一个外部的响应信号（此时 rxf = 1），接收到 ack 信号之后（意味着数据已经被读取）会将标记传送给占位符 PM1，并重置移位寄存器和 4 位计数器。

有效弧线在本例中的作用是同步两个佩特里网络，产生的米利输出信号 PD，使得网络根据不同条件形成不同的走向。

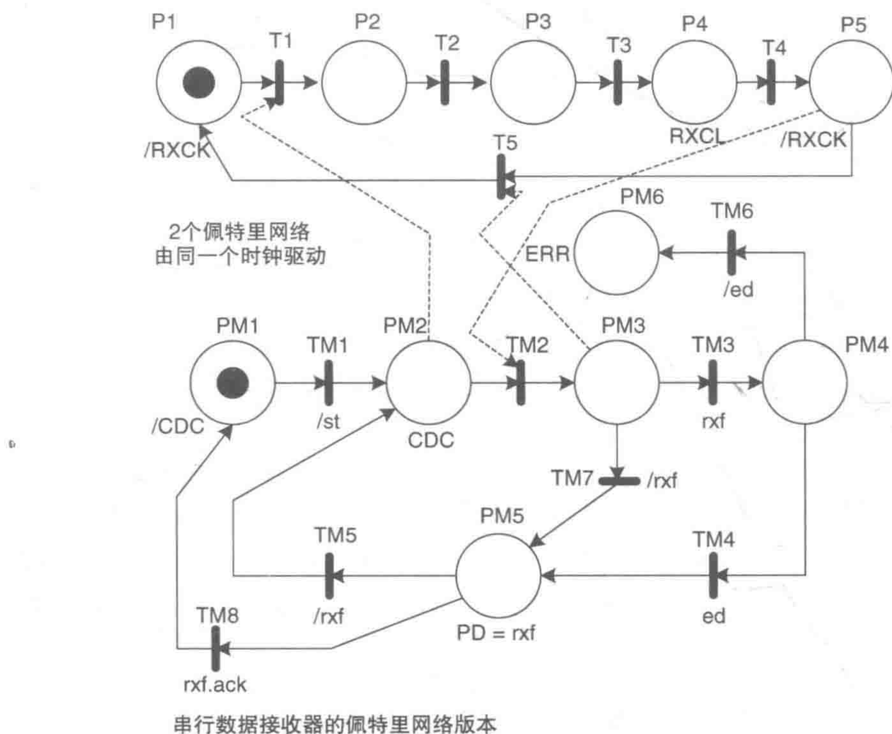


图 10.22 异步串行系统佩特里网络框图

10.7.1　第一个佩特里网络的公式

$$P1 = T5 + P1 \cdot /T1 \quad T1 = P1 \cdot PM2 \cdot /P2$$
$$P2 = T1 + P2 \cdot /T2 \quad T2 = P2 \cdot /P3$$
$$P3 = T2 + P3 \cdot /T3 \quad T3 = P3 \cdot /P4$$
$$P4 = T3 + P4 \cdot /T4 \quad T4 = P4 \cdot /P5$$
$$P5 = T4 + P5 \cdot /T5 \quad T5 = P5 \cdot PM3 \cdot /P1$$

10.7.2　第一个佩特里网络输出公式

$$RXCK = P4$$

10.7.3　主佩特里网络公式

$$PM1 = TM8 + PM1 \cdot /TM1$$
$$PM2 = TM5 + TM1 + PM2 \cdot /TM2$$
$$PM3 = TM2 + PM3 \cdot /TM3 \cdot /TM7$$
$$PM4 = TM3 + PM4 \cdot /TM4 \cdot /TM6$$
$$PM5 = TM4 + TM7 + PM5 \cdot /TM5 \cdot /TM8$$
$$PM6 = TM6 + PM6$$
$$TM1 = PM1 \cdot /st \cdot /PM2$$
$$TM2 = PM2 \cdot P5 \cdot /PM3$$
$$TM3 = PM3 \cdot rxf \cdot /PM4$$
$$TM4 = PM4 \cdot ed \cdot /PM5$$
$$TM5 = PM5 \cdot /rxf \cdot /PM2$$
$$TM6 = PM4 \cdot /ed \cdot /PM6$$
$$TM7 = PM3 \cdot /rxf \cdot /PM5$$
$$TM8 = PM5 \cdot rxf \cdot ack \cdot /PM1$$

10.7.4　主网络输出公式

CDC =/PM1 低有效

PD = PM5. rxf 米利高有效

ERR = PM6 高有效

图 10.23 给出了佩特里网络的仿真结果。仿真构建的测试平台可以模拟所有的佩特里网络路径。这需要对信号 rxf、ack 和 ed 的状态进行人为的控制。仔细研究图 10.23，便可以发现整个系统的测试路径。

基本上，仿真波形直观地反映了有效弧线控制移位寄存器时钟的产生（P1 ~ P5），以及主佩特里网络的运行情况（PM1 ~ PM6）。

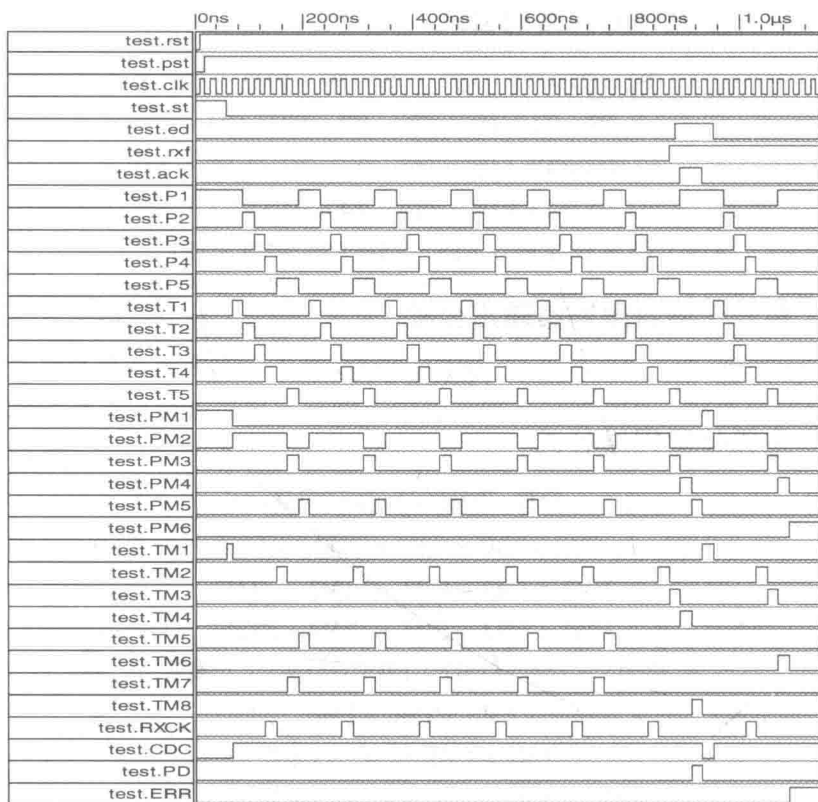


图 10.23 佩特里网络仿真波形图

图 10.24 是关于 RXCK 信号脉冲的进一步分析。根据图中所给出的信息，在串行数据接收过程中，系统每经过 7 个 FSM 时钟脉冲，会产生一个移位寄存器时钟脉冲。因此，对于 1×10^6bit/s 的波特率来说，FSM 的时钟频率应为 7MHz。

图 10.23 里也很清晰地反映了有效弧线的一系列状态变化。整个仿真在系统收到报错信号时结束，迫使佩特里网络进入占位符 PM6。

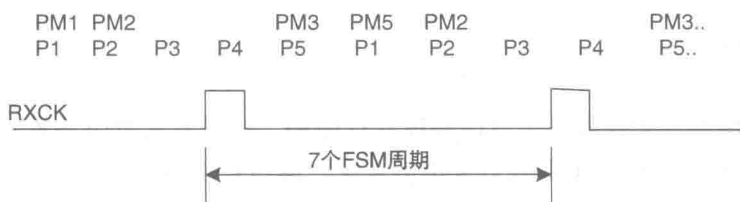
系统中的移位寄存器、4 位计数器、与门逻辑电路，以及数据锁存器等器件定义和连接，也是确保功能设计完备的关键。

10.7.5 移位寄存器

系统中用到的是 11 位移位寄存器。附录 B 中图 B.12a 和图 B.12b 有详描述。

10.7.6 移位寄存器的公式

对于一个常规的 m 位（D 触发器的数量）移位寄存器：



每个rxclk周期长度等同于7个FSM周期

波特率的换算 = FSM频率/7

在PM2，T1启动后P1 ~ P4进程也随之开始
到达P5时，TM2启动，全网络此时可以进入PM3，随后进入PM5(条件为rxf=0)然后回到PM2

RXCK信号在P4生成

整个过程可以一直持续到rxf为逻辑1(意味着整包数据接收完毕)，此时循环被打破。佩特里网络进入PM4，如果此时ed=1(无误码)数据将被加载到锁存器(pd=1)等待外部器件读取

参考图10.23

图 10.24　佩特里网络数据接收详细流程

$Q_0 \cdot d = din$　数据输入

$Q_n \cdot d = Q_{n-1}$

公式适用于标记从 $n=1$ 到 $n=m-1$ 的所有触发器，其中 m 表示移位寄存器里触发器的数量。

那么引申到 11 位移位寄存器，公式则变为：

$Q_0 \cdot d = rx$

$Q_n \cdot d = Q_{n-1}$，其中 n 的变化范围是从 1 到 $m-1$，$m=11$。

不过这里没有必要将移位寄存器的时钟信号 RXCK 引入公式，因为它的状态由佩特里网络来控制。

10.7.7　4 位计数器

它可以是异步二进制计数器也可以是同步的。附录 B 里图 B.13a 和图 B.13b 有详细说明。

10.7.8　数据锁存器

这是一个标准的拥有 8 个 D 触发器的并行数据锁存器，每一个触发器自带输入和输出端口，且通过数据锁存信号 PD 来控制。

奇偶校验逻辑电路也可以加入系统，原理和第 4 章所介绍的一样。

10.8 小结

佩特里网络在硬件设计中可以实现并行控制。本章对此部分内容进行了阐述，并向读者展示如何运用 HDL 设计类似的系统。运用有效/失效弧线可以实现并行佩特里网络的同步。

参考文献

1. Fernandes JM, Adamski M, Proeca AJ. VHDL generation from hierarchical Petri net specifications of parallel controllers. IEE Proc Comput Digital Technol 1997; 144(2): 127–135.

附　　录

附录 A　本书所使用的逻辑门和布尔代数

本章向读者介绍一些基本的布尔代数法则及其运用。下面的内容是建立在假设读者已经掌握这些法则的基础之上的。因此它们仅仅是用作参考。

A.1　本书涉及的基本逻辑门符号和布尔代数表达式

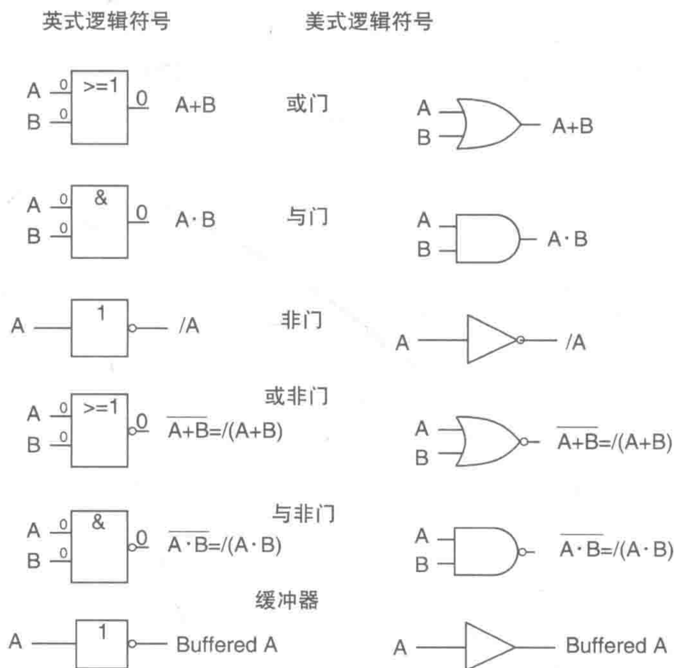


图 A.1　基本逻辑门

A.2　异或门和同或门

本书所介绍的逻辑系统中也用到了异或门和同或门。

英式逻辑门符号　　美式逻辑门符号

异或

A ^B

异或门是由与门、或门和非门共同构成
布尔公式为F=A·/B+/A·B

A	B	F	/F
0	0	0	1
1	0	1	0
0	1	1	0
1	1	0	1

/F为同或结果

/(A ^B)

英式同或门符号　　美式同或门符号

同或门的功能是将A和B进行比较

图 A.2　异或和同或逻辑门

A.3　布尔代数法则

我们用布尔逻辑公式和门电路来讲解这些法则。

A+1=1

A+0=A

A+A=A

A+/A=1

图 A.3　布尔代数：基本或法则

A. 3. 1　基本或法则

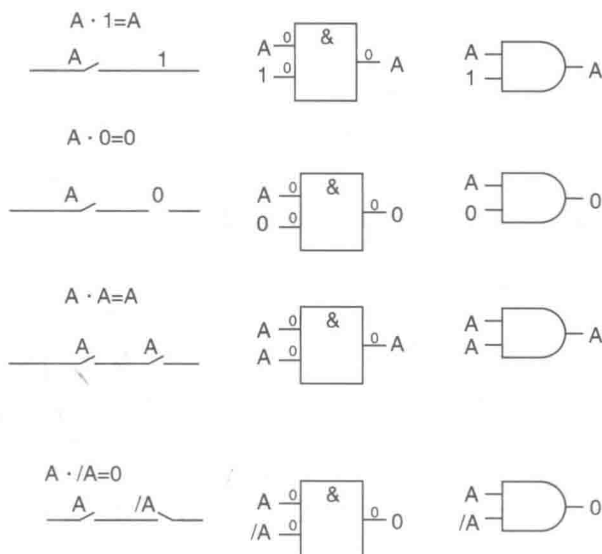
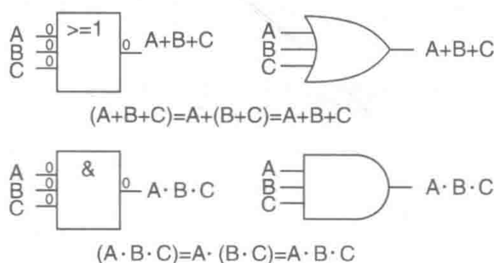


图 A. 4　布尔代数：基本与法则

A. 3. 2　基本与法则

结合律：



交换律：

A+B=B+A

A·B=B·A

这两个定律表明公式中变量的顺序(A、B和C)是可以
重新组合(结合律)和交换的(交换律)

图 A. 5　结合律和交换律

A.3.3 结合律和交换律

A.3.4 分配律

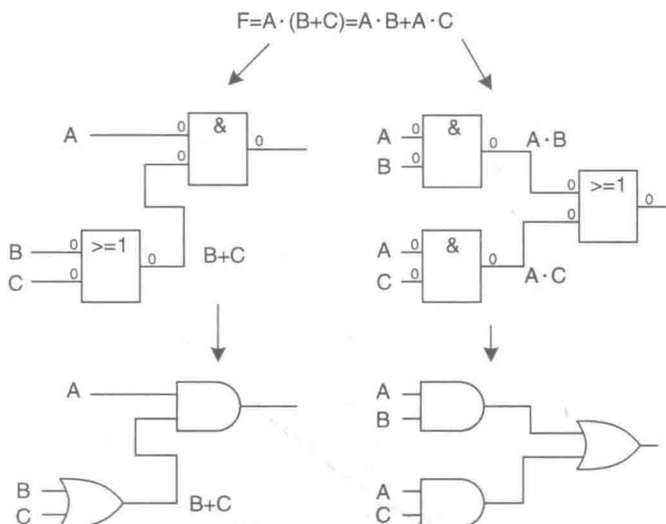


图 A.6 分配律

A.3.5 针对静态逻辑 1 竞争冒险的辅助法则

辅助法则（见图 A.7）在本书中被大量运用，且显得尤为重要。

辅助法则的证明

图 A.7 最后留下的两个问题中，第一个解答如下：

$$
\begin{aligned}
Y &= x + /x \cdot z \\
&= (x + /x) \cdot (x + z) \\
&= x \cdot x + x \cdot z + /x \cdot x + /x \cdot z \\
&= x + x \cdot z + 0 + /x \cdot z \\
&= x \cdot (1 + z) + /x \cdot z \\
&= x + /x \cdot z
\end{aligned}
$$

而图 A.7 中最后第 2 个问题$/p + (q \cdot p) = ?$ 的答案很明显是$/p + q$。大家可能会注意到辅助法则起始就是统一法则的一种特殊形式。

A.3.6 统一法则

请大家看下面的公式：

$$Y = x + /x \cdot w \cdot z$$

公式中，如果 $w = z = 1$，则

$$Y = x + /x \cdot 1 \cdot 1 = x + /x$$

辅助定律统一法则的特殊形式

A+(B C)=(A+B) (A+C) 和 (A+B) (A+C) = A+(B C)

右边可以看作是电路的简化

不过，下面是一个更加有趣的现象

A + (/A·B) = (A+/A)·(A+B) =1·(A+B) = A+B

在B=1时，上述公式给出简化的过程并清除了静态1冒险

(A+/A)·(A+1) = (A+/A)

Try x+(/x·z) = ?

And/p + (q·p) = ?

图 A.7　辅助法则

结果就是 1。

然而，对于门电路来说，某些时候会出现器件的延迟，而导致某个参数的状态仍为逻辑 0（这就是所谓的静态 1 冒险）。为了避免这种情况，公式里可以添加一个补偿项，用 w 和 x 的乘积来表示，确保最终 Y 的值保持在逻辑 1。

$$Y = x + /x \cdot w \cdot z + w \cdot z$$

因此当 w 和 z 均为 1 时，$Y = x + /x + 1$，多出来的 1 对于 $x + /x$ 是一种补偿，防止出现冒险。

现在我们来看下面的公式：$Y = x + /x \cdot w$，添加补偿项后变成：

$$\begin{aligned} Y &= x + /x \cdot w + w \\ &= x + w \cdot (/x + 1) \\ &= x + w \end{aligned}$$

因此在这里用统一法则对公式 $Y = x + /x \cdot w$ 进行优化后得到 $Y = x + w$，和用辅助定律的结果是一样的。

事实上，辅助定律就是统一法则的一种特殊情形，也是统一法则的一种补充。

再看一个例子（运用辅助定律）：

$$\begin{aligned} Y &= /x + x \cdot z \\ &= /x + z \end{aligned}$$

推导过程是：

$$
\begin{aligned}
/x + x \cdot z &= (/x + x) \cdot (/x + z) \\
&= 1 \cdot (/x + z) \\
&= /x + z
\end{aligned}
$$

下面给出的两个例子中，优化掉的参数用横线划去表示~~/R~~。

$$
\begin{aligned}
P &= q \cdot r + q \cdot /r \cdot s \\
&= q \cdot (r + /r \cdot s) \\
&= q \cdot (r + \not{/r} \cdot s) \\
&= q \cdot (r + s) \\
&= q \cdot r + q \cdot s \\
Y &= s \cdot t \cdot /x + s \cdot t \cdot x \cdot z \\
&= s \cdot t \cdot (/x + x \cdot z) \\
&= s \cdot t \cdot (/x + z)
\end{aligned}
$$

因此 $Y = s \cdot t \cdot /x + s \cdot t \cdot z$

要注意的是优化掉的是系统的输入信号。对于某些情况来说是成立的，但是当用 PLD 器件时，它往往提供的是由与门阵列组成的输入端口，这时候上述优化就不一定成立了。

不过，还是要记住，用辅助定律可以避免潜在的静态 1 冒险。对于上述的 P，就是 r + /r；对于上述的 Y，就是 x + /x。

A.3.7　逻辑门里信号的延迟效应

在本书第 3 章、第 4 章和第 9 章，读者们都可以找到信号的延迟，对电路行为构成影响的例子。图 A.8 给出了一个信号延迟给门电路造成影响的情况，对于不同的逻辑门，两个输入信号 A 和/A 之间的延迟，能够在输出端清晰地反映出来。

对于与门（或者与非门），延迟造成两个信号重叠的部分都为逻辑 1 时，使得逻辑门的输出产生了变化。

对于或门（或者或非门），延迟造成两个信号重叠的部分都为逻辑 0 时，使得逻辑门的输出产生了变化。

这里输出端所产生的变化就是我们不希望看见的毛刺。这种所谓的“毛刺”会影响 FSM 的误操作。它们通常会在系统中有两个信号（不一定是如图 A.8 所示的正反两个信号）同时改变状态时出现。这会导致 FSM 中两个二次状态变量发生变化；比如，当没有用单位距离编码进行 FSM 设计时，就有可能发生这种情况。

由于逻辑门之间的延迟效应，这种情况在逻辑电路内部，出现两个信号发生变化时会自然显现。大部分介绍高级数字系统设计的书籍，都会提到这样的问题并给出一系列的解决方案。这里本书将不做更加深入的讨论。

A. 3. 8　De Morgan 法则

De Morgan 法则在本书第 9 章被大量运用。

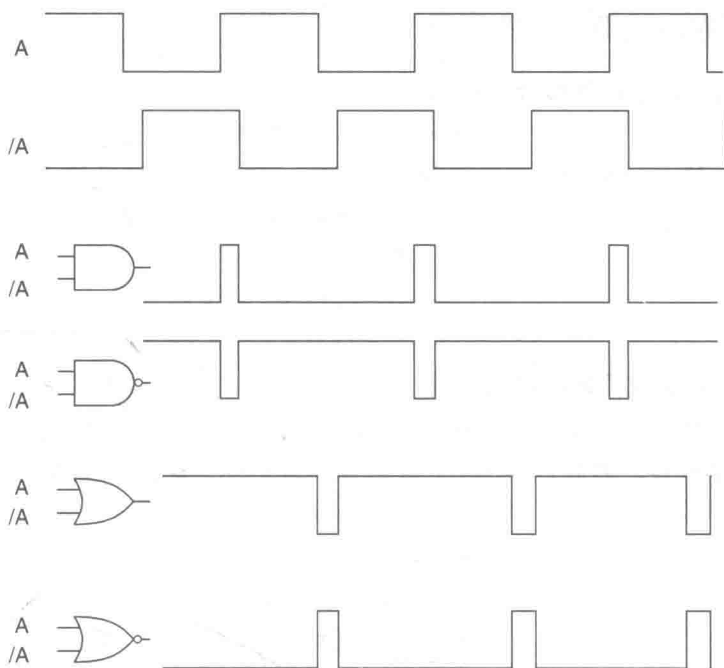
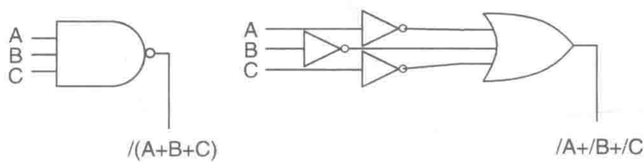


图 A. 8　逻辑门输出信号延迟的影响

De Morgan 法则在第9章被大量应用

/(A·B·C) = /A + /B + /C



/(A + B + C) = /A·/B·/C

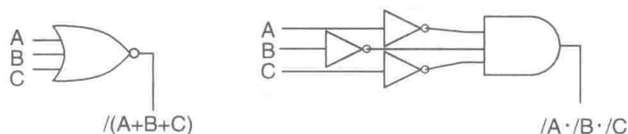


图 A. 9　De Morgan 法则

De Morgan 法则常用来将与非门（NAND），即/（a·b），转换成或非门（NOR），即/a+/b。也可以将或非门，即/（a+b），转换成与非门，即/a·/b。

A.4 运用布尔代数的一些例子

A.4.1 将与门和或门转换成与非门

$$Z = x \cdot y + /x \cdot /y$$

运用 De Morgan 理论，可以得到/(A+B) = /A·/B，这里的 A 和 B 可以用任何乘积项来代替。因此，本例中，A 被替换为 x·y，B 被替换为/x·/y。

$$\begin{aligned} Z &= //Z \\ &= //(x \cdot y + /x \cdot /y) \\ &= /(/(x \cdot y) \cdot /(/x \cdot /y)) \end{aligned}$$

A.4.2 将与门和或门转换成或非门

运用 De Morgan 理论，可以得到/（A·B）=/A+/B，这里的 A 和 B 可以用任何乘积项来代替。因此，本例中，A 被替换为 x·y，B 被替换为/x·/y。

$$\begin{aligned} Z &= x \cdot y + /x \cdot /y \\ &= //Z \\ &= //(x \cdot y + /x \cdot /y) \\ &= //(//(x \cdot y) + //(/x \cdot /y)) \\ &= //(/(/x + /y) + /(x + y)) \end{aligned}$$

A.4.3 逻辑相邻定律

在卡诺图中，运用此定律可以有效简化布尔代数公式，例如/b+b=1；因此对于公式：

$$\begin{aligned} F &= b \cdot c + b \cdot /c \\ &= b \cdot (c + /c) \\ &= b \cdot 1 \\ &= b \end{aligned}$$

这样就将 c 简化掉了。

再来看一个例子：

$$\begin{aligned} X &= a \cdot /b \cdot c + a \cdot b \cdot c \\ &= a \cdot c \cdot (/b + b) \\ &= a \cdot c \cdot 1 \\ &= a \cdot c \end{aligned}$$

下面这个例子使用了两次相邻定律：

$$\begin{aligned} Q &= /a \cdot /b \cdot /c + /a \cdot /b \cdot c + a \cdot b \cdot /c + a \cdot /b \cdot /c \\ &= /a \cdot /b \cdot (/c + c) + a \cdot /c \cdot (b + /b) \\ &= /a \cdot /b + a \cdot /c \end{aligned}$$

逻辑相邻定律在设计同步 FSM 时用得比较普遍。在第 9 章中建立异步 FSM 正确的运作方式时，也发挥了作用。

辅助法则和相邻定律，在简化 D 触发器和 T 触发器公式时都经常被运用。一个典型的状态表公式如下：

$$
\begin{aligned}
A \cdot d &= /A \cdot B \cdot st + A \cdot B + A \cdot /B \cdot sp \\
&= B \cdot st + A \cdot B + A \cdot sp
\end{aligned}
$$

推导过程如下：

$$
\begin{aligned}
A \cdot d &= B \cdot (/A \cdot st + A) + A \cdot (B + /B \cdot sp) \\
&= B \cdot (st + A) + A \cdot (B + sp) \\
&= B \cdot st + AB + AB + A \cdot sp
\end{aligned}
$$

由于 $A \cdot B + A \cdot B = A \cdot B$

因此 $A \cdot d = B \cdot st + A \cdot B + A \cdot sp$

A.5　小结

本章主要归纳了布尔代数的一些基本定律，并讨论了在本书中运用到的一些技巧。它们给那些平时不太用到布尔代数的读者们提供了一个参考。更多关于布尔代数的内容可以在其他大部分讲述数字逻辑设计的书中找到。

附录 B　计数器和移位寄存器电路设计方法

本附录涵盖了一些设计同步二进制计数器和移位寄存器的方法和技巧。这些方法在前面的一些章节里已经得到了运用。

B.1　同步二进制递增或递减计数器

同步二进制递增或递减计数器，可以用来构建通用 n 比特二进制计数器。而且可以直接在 PLD、CPLD 和 FPGA 等器件里实现。为了直观地体现这个特性，这里拿一个 4 位递减计数器来举例。

表 B.1 是一个递减计数器的真值表，Q0 是最低位。设定它为同步计数器，因此系统中所有的触发器都由同一个时钟驱动。并且系统中使用的触发器类型是 T 触发器。大部分 CPLD 和 FPGA 器件都直接支持 T 触发器，或者可以用 D 触发器加一个异或输入来构建。

T 触发器的输入公式可以通过参照表 B.1，并加入每一个 0 到 1 和 1 到 0 转换所需的乘积项来获得。例如，根据表 B.1，触发器 q0 · t 的公式为

$$q0 \cdot t = s15 + s14 + s13 + s12 + s11 + s10 + s9 + s8 + s7 + s6 + s5 + s4 + s3 + s2 + s1 + s0 = 1$$

每一个 T 触发器需要变化状态（0 到 1 或者 1 到 0）的参数都被列入公式。

类似的公式可以最终用 Q0Q1Q2Q3 输出信号来表达，或者用图 B.1 的卡诺图

来呈现。使用图 B.1 可以大幅度简化触发器公式。

对于公式 q0 · t 来说，既然所有项都需要纳入考虑（每一个都赋值为 1），那么 Q0 触发器的 T 输入则为逻辑 1。

触发器 q1 · t 的公式为：

$$q1 \cdot t = s14 + s12 + s10 + s8 + s6 + s4 + s2 + s0 = /Q0$$

表 B.1　递减计数器

Q0	Q1	Q2	Q3	状态
1	1	1	1	s15
0	1	1	1	s14
1	0	1	1	s13
0	0	1	1	s12
1	1	0	1	s11
0	1	0	1	s10
1	0	0	1	s9
0	0	0	1	s8
1	1	1	0	s7
0	1	1	0	s6
1	0	1	0	s5
0	0	1	0	s4
1	1	0	0	s3
0	1	0	0	s2
1	0	0	0	s1
0	0	0	0	s0

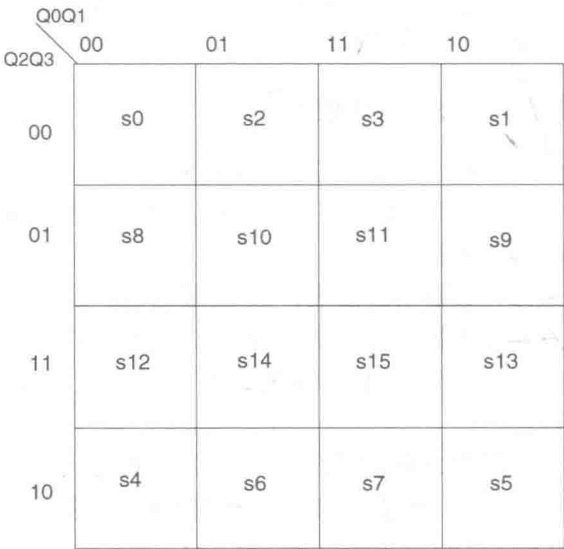


图 B.1　计数器状态卡诺图

从卡诺图来看，q1 · t 必须简化为/q0，因为 s14、s12、s10、s8、s6、s4、s2 和 s0 都含有一个 1。根据这种方法，q2 · t 和 q3 · t 的公式分别为 q2 · t = s12 + s8 + s4 + s0 = /Q0 · /Q1，q3 · t = s8 + s0 = /Q0 · /Q1 · /Q2。

由此可以得出触发器的公式遵循如下的格式：

$$qx \cdot t = /Q(x-1) \cdot /Q(x-2) \cdot /Q(x-3) \cdot \cdots \cdot Q(x-x) \quad (B.1)$$

式（B.1）描述的是用 T 触发器构建的递减计数器的 p 项。对于每个触发器来说，这样的公式可以直接输入到 Verilog HDL 文件中去用作代码。

对于递增计数器，可以用 q 项来替代式（B.1）里的/q 项：

$$qx \cdot t = Q(x-1) \cdot Q(x-2) \cdot Q(x-3) \cdot \cdots \cdot Q(x-x) \quad (B.2)$$

或者通常的表达方法是：

$$qn \cdot t = \prod_{p=1}^{p=n} Q(n-p) \quad (B.3a)$$

条件是：

$$q0 \cdot t = 1 \quad (B.3b)$$

对于每个触发器来说，∏符号代表所有输出信号的乘积（即与门）。注意 T 触发器 Q0 的输入是逻辑 1，这没有在式（B.3a）中体现。

运用卡诺图（见图 B.1）的方法，也能得出与递减计数器类似的结果，只是计数的方向是相反的而已。

B.2　用 T 触发器构建 4 位同步递增计数器

图 B.2 是一款 4 位递增同步计数器，用上一节介绍的 T 触发器来完成。

每个 T 触发器对应的公式为：

$$Q0 \cdot t = 1$$

$$Q1 \cdot t = Q0$$

$$Q2 \cdot t = Q0 \cdot Q1$$

$$Q3 \cdot t = Q0 \cdot Q1 \cdot Q2$$

代码 B.1 是模块的 Verilog HDL 描述：

```
//4 位计数器
//定义 T 触发器
module T_ FF (q, t, clk, rst);
output q;
input t, clk, rst;
reg q; //输出信号 q 必须是寄存器型
always @  (posedge clk or negedge rst)
   if (rst  = = 0)
     q  < = 1'b0;
   else
     q  < =  t^q; //T 触发器用异或门来运算
```

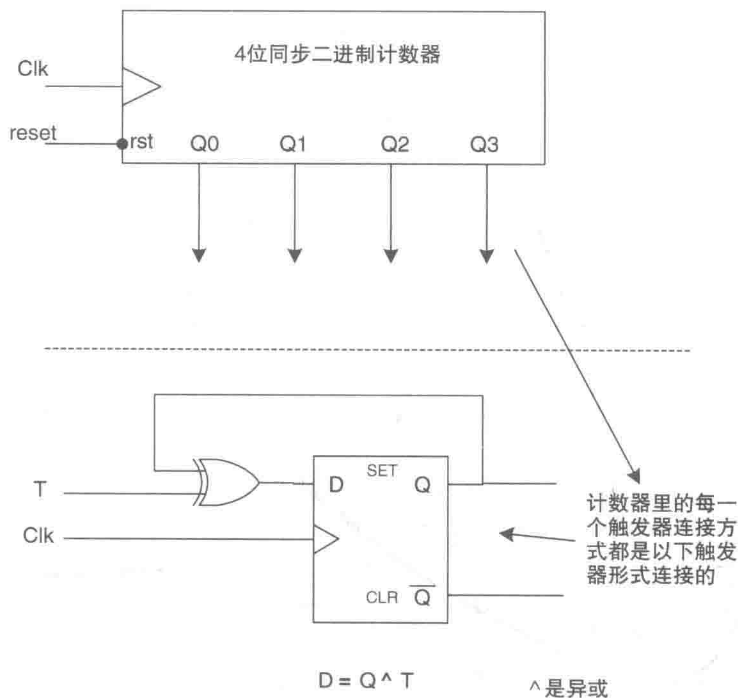


图 B.2　4 位同步计数器模块框图

```
endmodule

//定义计数器
module counter (Q0, Q1, Q2, Q3, clk, rst);
input clk, rst; //clk 和 rst 信号是输入
output Q0, Q1, Q2, Q3; //输出 Q
wire t0, t1, t2, t3; //所有的输入信号 t 都是内部连接线网

//定义每一个 T 触发器
T_ FF ff0 (Q0, t0, clk, rst);
T_ FF ff1 (Q1, t1, clk, rst);
T_ FF ff2 (Q2, t2, clk, rst);
T_ FF ff3 (Q3, t3, clk, rst);
//定义每一个输入信号 t 的逻辑关系
//我们用连续赋值语句来表达
```

```
assign

t0 =1'b1, //每一个线网都根据计数器的设计要求赋值
t1 =Q0,
t2 =Q0 & Q1, //与门
t3 =Q0 & Q1 & Q2;

endmodule //定义模块结束

//测试代码
module test;
reg clk, rst; // 两个输入必须为寄存器型
//由于测试过程中不需要线网型变量，而且此时仿真时计数器没有任何外部连接
counter count (Q0, Q1, Q2, Q3, clk, rst);
initial
begin
    $dumpfile ("counter4.vcd"); //波形文件
    $dumpvars ; //将所有赋值加载到文件里
   rst =0; //复位信号将电路初始化
   clk =0; //将时钟设为低
     #10rst =1; //延迟 10 个时钟单位以后，将复位信号释放
   repeat (17)
     #10  clk = ~clk; //每隔 10 个时钟单位将时钟反转，一共 17 次
     #20 $finish ; //20 个时钟单位之后结束仿真
end //结束测试代码
endmodule //测试模块结束
```

代码 B.1 计时器模块语言描述和测试平台代码

代码 B.1 是完整的 Verilog 模块描述和测试代码。其中包含了 T 触发器的构建（用行为模式来定义）。

之后利用定义的 T 触发器和表达触发器之间关系的连续赋值语句，来完成计数器的模块定义。注意：这里模块输入输出的定义，被放在模块信号列表的外部。

测试模块的代码在模块代码后面。其中包含 4 位计数器和测试步骤。两个 $ 命令用来保存图 B.3 所对应的波形图，这样仿真结果可以保存到 word 文档里便于打印。命令 $dumpfile（“counter. vcd”）用来给文件命名。命令 $dumpvars；的作用是将仿真结果加载到文件中。

最终结果被存为图元文件。至于仿真结果是否符合预期，大家可以从图中看得很明显。

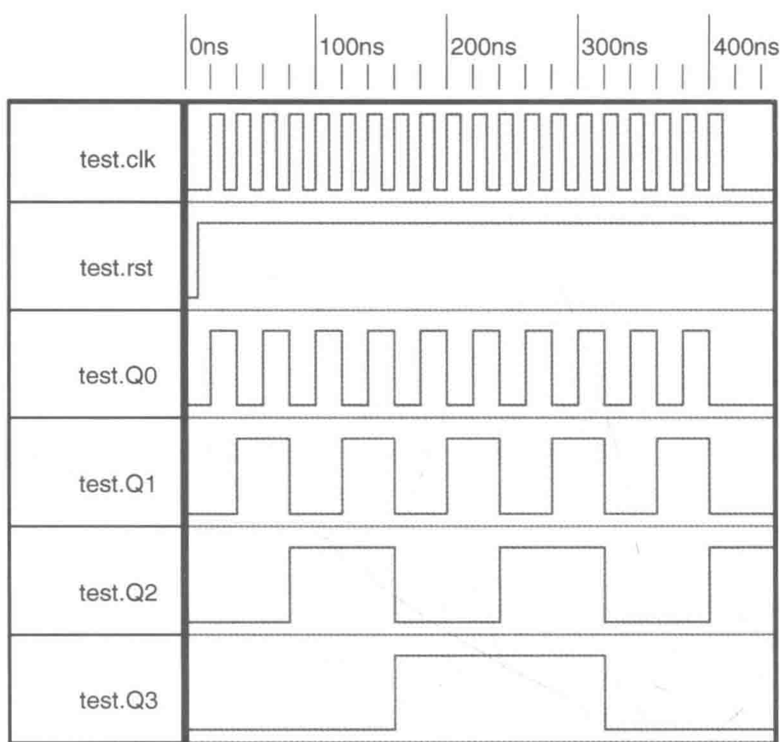


图 B.3 4 位计数器仿真结果

B.3 并行加载计数器：运用 T 触发器

对于用廉价的 PLD 器件构建的并行加载计数器，如果系统中没有异步预设或者复位信号作为触发器的输入，那么就必须加入同步的并行输入信号。实现的方法是在公式 qx · t 里添加额外的乘积项。

式（B.4）是带有额外输入的单个 T 触发器：

$$qx \cdot t = ptermx \cdot /load + px \cdot /Qx \cdot load + /px \cdot Qx \cdot load \quad (B.4)$$

其中的输入信号 load 用来把并行数据以同步的方式加载到 T 触发器中。此时，load 是高有效的。

公式中的 ptermx · /load 是一般计数器都要用到的乘积项，且在 load 输入没有被激活时有效。px · /Qx · load 是激活触发器的并行输入项，/px · Qx · load 用来复位触发器。

图 B.4 是单个触发器的大致的构架。其他触发器都可以采用类似的构架。这里假设输入信号 load 是高有效。因此，在计数器工作时，load 信号应该为低（逻辑 0）。

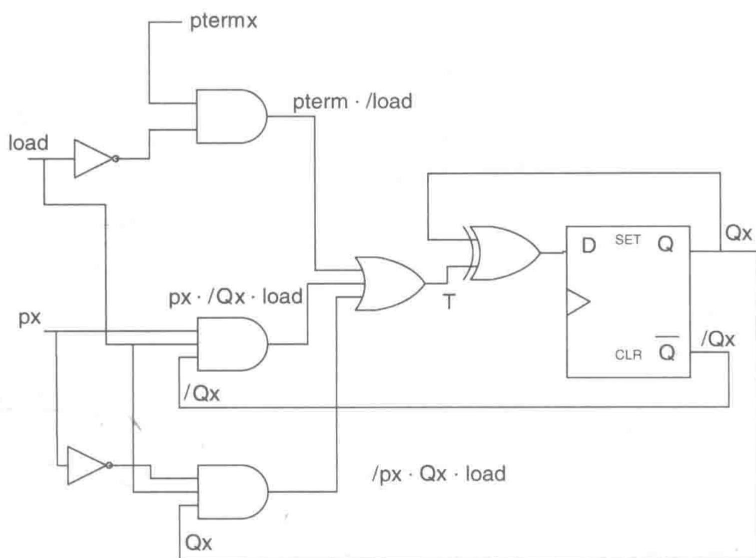


图 B.4 并行加载计数的单个触发器单元

并行加载递增/递减计数器可以用式（B.1）、式（B.2）和式（B.4）来生成，它们应用范围广，包括存储芯片的地址解码单元。

因此，对于存储芯片的控制来说，不单单只有时序控制，采用并行输入还能够随时对存储芯片进行全方位的控制。

B.4 在低成本 PLD 器件平台上用 D 触发器来构建并行加载计数器

上述设计中的 T 触发器也可以用 D 触发器来替代，而且不需要初始化和复位输入信号。对于许多低成本的 PLD 器件来说，没有异步初始化和复位信号的 D 触发器方案更加受欢迎。

请大家看图 B.5，单个触发器的公式如下：

$$qx \cdot d = px \cdot /l + pterm \cdot l \tag{B.5}$$

其中 l 是并行加载输入，/l 是将其取反。公式定义了构成计数器的每一个触发器的基本格式。

乘积项 pterm 的值取决于计数器的运行方式。而且也没有一个通用的方法。因此相比于 T 触发器的方案，这个方法其实并不省心。

这里可以举一个简单的 3 位同步二进制递增计数器来说明。

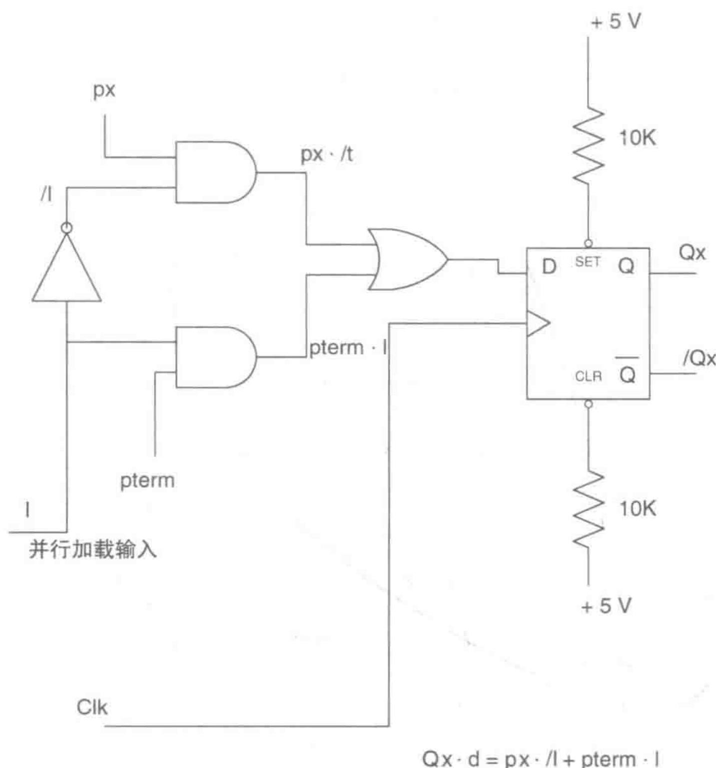


图 B. 5 同步并行加载计数器单个 D 触发器构架

B. 5 二进制递增计数器：带有并行输入

为了阐述硬件电路的构成，图 B. 6 给出了一个简单的 3 位并行加载二进制计数器。

图中的状态序列对应二进制计数的序列。状态图（卡诺图）用来帮助简化公式里的 pterms 项（以更加简洁的方式体现）。还有就是所有 D 触发器的输入公式。

和用 T 触发器设计同步并行加载计数器相比，这种方法需要明确每一个触发器的 pterm 项。总体来说，这种方法缺乏系统性，必须把每一个触发器的情况都要考虑一遍。

然而，用 D 触发器的一个好处，在于计数序列没有必要一定是纯二进制计数（即也可以用单位距离编码序列进行设计）。

当然，计数器的行为可以直接用 Verilog HDL 来描述，而且通常这样的方法也更常见。因此上述方法只是让大家对布尔代数公式在计数器中的应用有更加深入的认识。

Q0	Q1	Q2	状态	
0	0	0	s0	
1	0	0	s1	
0	1	0	s2	
1	1	0	s3	
0	0	1	s4	状态序列
1	0	1	s5	
0	1	1	s6	
1	1	1	s7	

Q2 \ Q0 Q1	00	01	11	10	
0	s0	s2	s3	s1	状态图
1	s4	s6	s7	s5	

pterms

q0 · d = /Q0

q1 · d = Q0 · /Q1 + /Q0 · Q1

q2 · d = Q2 · /Q1 + Q2 · /Q0 + /Q2 · Q1 · Q0

q0 · d = p0 · /I+ (/Q0) · I

q1 · d + p1 · /I+ (Q0 · /Q1 + /Q0 · Q1) · I

q2 · d + p2 · /I+ (Q2 · /Q1 + Q2 · /Q0 + /Q2 · Q1 · Q0) · I

带有并行加载输入的完整公式

图 B.6　带有并行输入的 3 位二进制同步计数器公式推导过程

B.6　驱动计数器（包括 FSM）的时钟电路

晶振的电路设计方法多种多样，不过图 B.7 是一种比较常用的方案。这里提到时钟电路是为了将设计二进制计数器的方方面面都覆盖到。

图 B.7 中提供的电路方案通过电容 C1 和 C2 进行谐波抑制，特定的电容值目的在于降低容抗。

B.7　使用自由状态设计计数器

在用 FSM 设计计数器时，有时候并不是每一个状态都和计数序列一一对应。这种设计方法可以减少一些逻辑门的数量。

图 B.8 是一个旋转环形计数器的例子，之所以起这个名字，是由于模块中所有触发器的连接方式好比一个圆环，并且首位两个触发器之间是相连的。图中是状态序列和带有自由状态的卡诺图。

状态序列仅仅列出了计数所需的状态表，从中可以看出，状态 s2、s4、s5、s6、s9、s10、s11 和 s13 不在计数序列之内，所以它们都是自由态，用 X 来标记。

根据计数序列和状态图，对应每一列 0 到 1 和 1 到 1 的状态转换，可以推导出

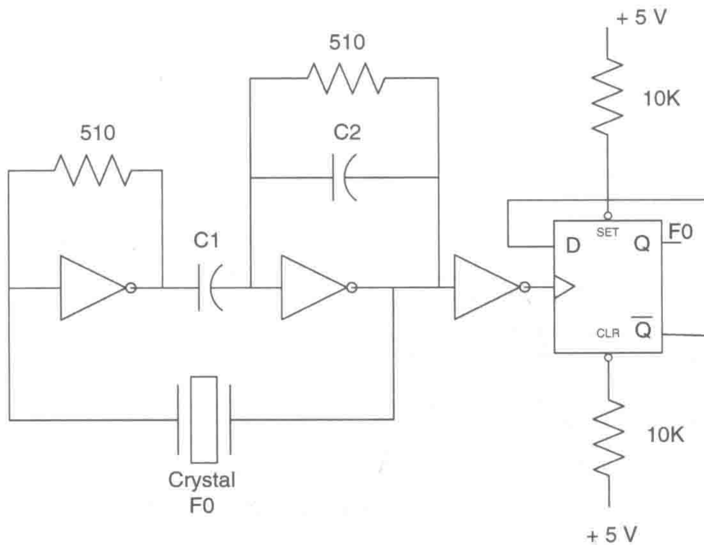


图 B.7 典型晶振电路

模块中每一个 D 触发器对应输入（Qx · d）的公式。自由态加在每一个公式末尾。公式最终简化的依据是状态图。

举例来说，公式 Q0 · d 由状态 s0、s1、s3 和 s7 与自由态 s2、s4、s5、s6 组成，最终简化为/Q3（图 B.8 中虚线已经标出）。其余的公式也是遵循这个规则。

B.8 移位寄存器

移位寄存器可以看作是同步计数器的一种特殊形式。通常来说，并行加载移位寄存器在设计中应用的场合较多（参考第 4 章设计实例）。并行加载移位寄存器每一位的公式如下：

$$Q0 \cdot d = din \cdot ld + p0 \cdot /ld \quad \text{(B.6a)}$$

$$Qx \cdot d = Q(x-1) \cdot ld + px \cdot /ld \quad \text{(B.6b)}$$

这里加载输入信号 ld 是低有效，din 是输入数据。

如果串行输入数据为 0，即 din = 0，那么移位寄存器的设计会使用 D 触发器。

上述公式还可以用来构建 4 位并行加载移位寄存器，即：

$$Q0 \cdot d = din \cdot ld + p0 \cdot /ld \quad \text{(B.7)}$$

$$Q1 \cdot d = Q0 \cdot ld + p1 \cdot /ld \quad \text{(B.8)}$$

$$Q2 \cdot d = Q1 \cdot ld + p2 \cdot /ld \quad \text{(B.9)}$$

$$Q3 \cdot d = Q2 \cdot ld + p3 \cdot /ld \quad \text{(B.10)}$$

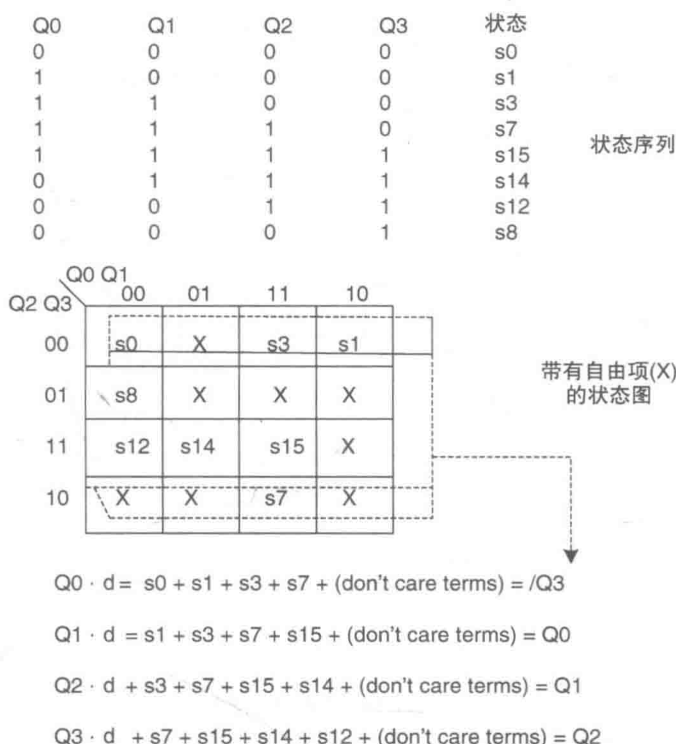


图 B.8　使用自由状态设计的旋转环形计数器

$$Sft_\ clk = clk \cdot ld \tag{B.11}$$

式（B.7）的第一项是串行数据输入，而式（B.8）~式（B.10）中，每个公式的第一项代表前一个触发器的输出，作为下一个触发器的数据输入（即标准的移位寄存器级联）。式（B.11）定义了移位时钟。时钟在加载过程中是不输出脉冲的。

图 B.9 是前面几个公式对应的移位寄存器的电路图。实际应用中，公式会直接用 Verilog HDL 来代替。如果将公式转换为 Verilog HDL，可以得到如下代码：

```
Q0d=din & ld | p0 & ~ld;
Q1d=Q0 & ld | p1 & ~ld;
Q2d=Q1 & ld | p2 & ~ld;
Q3d=Q2 & ld | p3 & ~ld;
Sft_ clk=clk & ld;
```

上述移位寄存器对应的代码一旦全部完成，就可以进行仿真并验证其功能。图 B.10 给出了仿真结果。

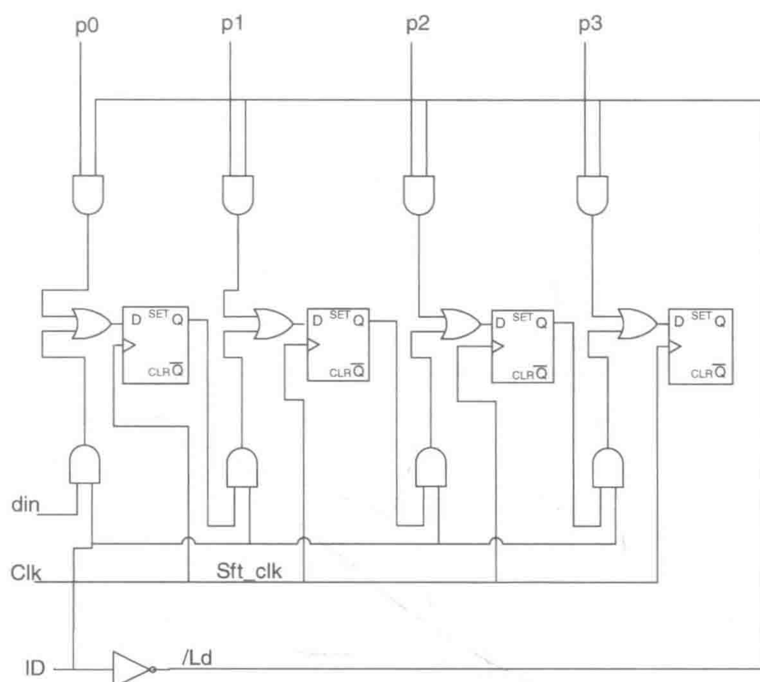


图 B.9 式（B.7）~式（B.11）对应的 4 位移位寄存器

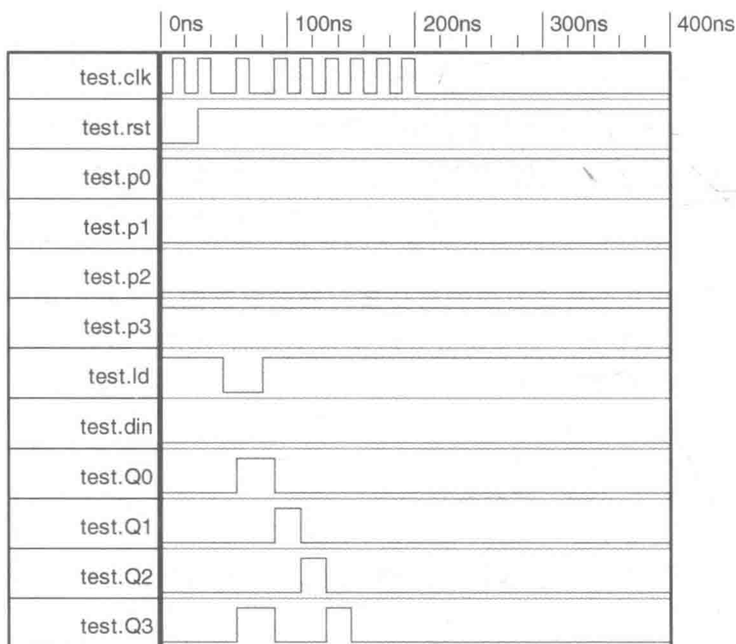


图 B.10 输入数据 din = 0 时 4 位移位寄存器仿真结果

在第 4 章，介绍过一个异步串行接收系统，其中用到移位寄存器接收数据，并将它们送到数据锁存器。此外，还有一个 4 位计数器是用来统计接收到的二进制数据的位数，并在一包完整数据接收完毕的时候通知 FSM（移位寄存器存满）。

下面将这两个模块完整的描述呈现给大家。

B.9　第 4 章里的异步接收器

图 B.11（这里借用图 4.21）是接收机的系统构成框图，其中包含了所需的各种不同模块。

FSM 的描述在 4.7 节中已经讲得很细致，图 4.22 是对应的状态图。

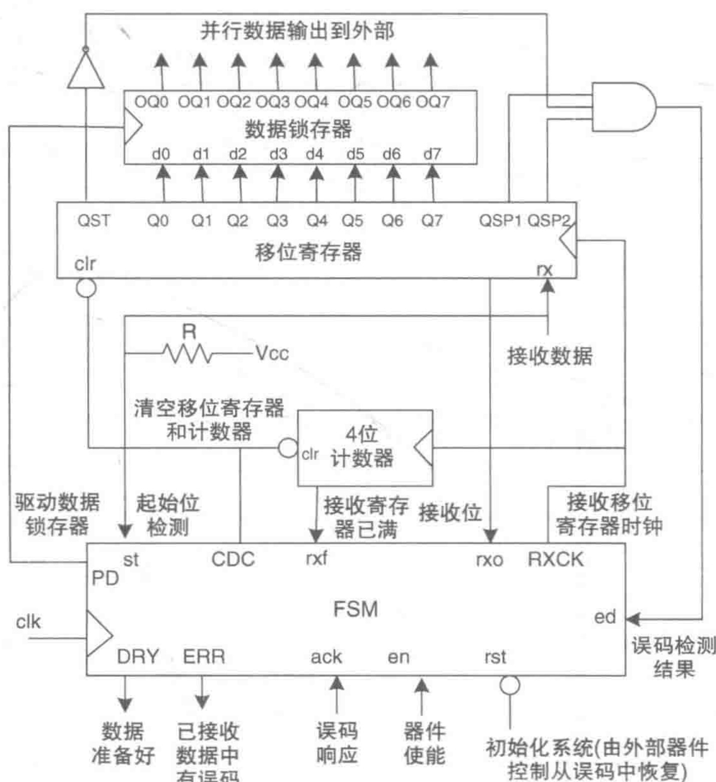


图 B.11　第 4 章的异步接收器

B.9.1　异步接收器中用到的 11 位移位寄存器

11 位移位寄存器含有一个起始位，8 位数据位（d0 ~ d7），以及两个结束位（sp1 和 sp2）。

输入数据 din 首先连接到 sp2 的触发器，然后移位进入 sp1 触发器。最后一个触发器是开始位的触发器，因为这个触发器对应进入移位寄存器第一位数据。图

B. 12a 描绘了其移位寄存的构架。

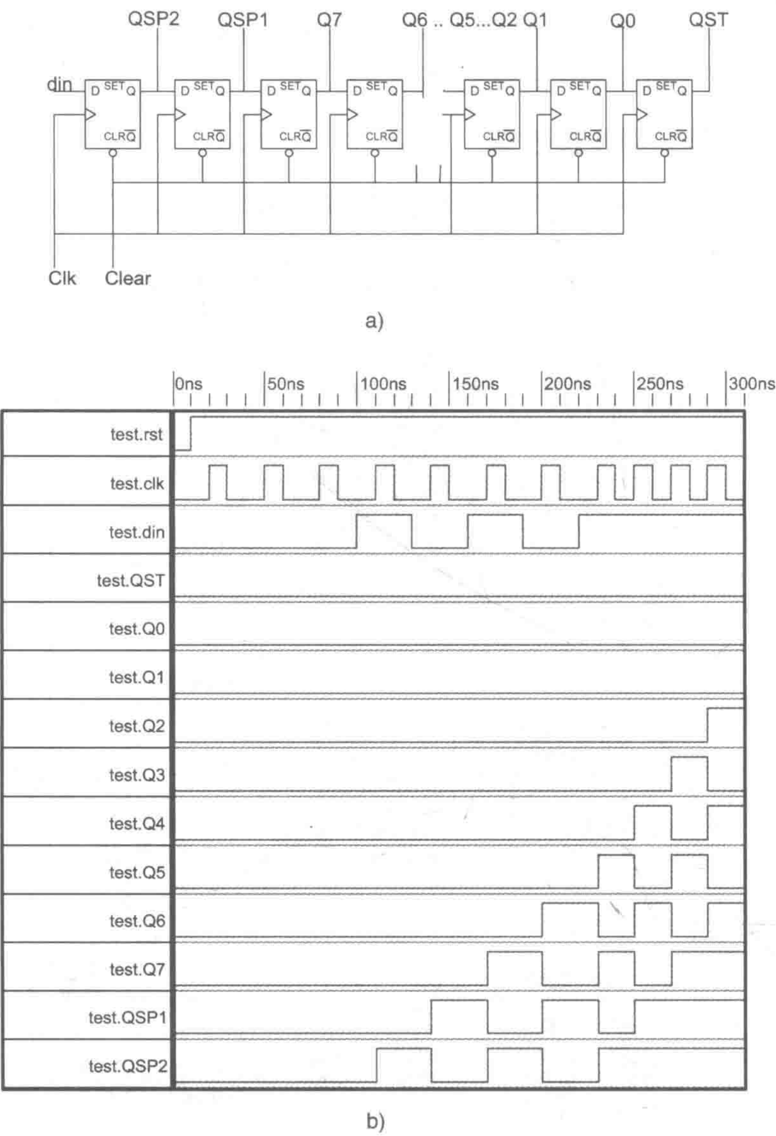


图 B. 12　a）移位寄存器电路 b）移位寄存器模块仿真

代码 B. 2 对应移位寄存器的 Verilog 源代码

```
//定义 D 触发器
module D_ FF (q, d, clk , rst);
output q;
input d, clk, rst;
```

```
reg q;
always @ (posedge clk or negedge rst)
    if (rst = =0)
      q< =1'b0;
    else
  q< =d;
endmodule
```

代码 B.2　移位寄存器里用到的 D 触发器 Verilog 语言描述

代码 B.3 是构建上述移位寄存器的代码

```
//-----------------------------------------
-----------------------------
//定义移位寄存器
//移位寄存器的时钟信号为 rxclk
//它受 FSM 控制
//数据包里协议位（st、sp1、sp2）拥有各自的触发器
//-----------------------------------------
-----------------------------
module shifter (rst, clk, din, QST, Q0, Q1, Q2, Q3, Q4, Q5, Q6, Q7,
QSP1, QSP2);
input clk, rst, din;
output QST, Q0, Q1, Q2, Q3, Q4, Q5, Q6, Q7, QSP1, QSP2;
wire dst, d0, d1, d2, d3, d4, d5, d6, d7, dsp1, dsp2;

  D_ FF_ qstd (QST, dst, clk, rst);
  D_ FF q0d (Q0, d0, clk, rst);
  D_ FF q1d (Q1, d1, clk, rst);
  D_ FF q2d (Q2, d2, clk, rst);
  D_ FF q3d (Q3, d3, clk, rst);
  D_ FF q4d (Q4, d4, clk, rst);
  D_ FF q5d (Q5, d5, clk, rst);
  D_ FF q6d (Q6, d6, clk, rst);
  D_ FF q7d (Q7, d7, clk, rst);
  D_ FF qsp1d (QSP1, dsp1, clk, rst);
  D_ FF qsp2d (QSP2, dsp2, clk, rst);
assign
  //这里要注意触发器之间的连接方式
dst =Q0,
d0 =Q1,
d1  =Q2,
```

```
  d2 =Q3,
  d3 =Q4,
  d4 =Q5,
  d5 =Q6,
  d6 =Q7,
  d7 =QSP1,
  dsp1 =QSP2,
  dsp2 =din;
endmodule
```

代码 B.3　移位寄存器测试固件代码

图 B.12b 是仿真结果，波形图显示其可以正常工作。

图中仿真时间到达 300ns 时，移位寄存器已经存满数据，并判断协议位数据符合条件。

B.9.2　4 位计数器

这里的计数器属于同步二进制递增计数器，当它数到十进制 11（二进制 1101）时会停止计数。计数器的输出是 RXF 信号。此信号在第 11 个时钟脉冲到达之后被拉高。

图 B.13a 是 4 位计数器的电路图。它由 4 个 T 触发器（通过用 D 触发器加上异或门反馈来构建）组成。其中 4 个与非门的作用是当计数器计到 11（Q3Q2Q1Q0 = 1011）时将模块停止。复位输入信号的作用是将计数器清零。

代码 B.4 是计数器对应的 Verilog 语言描述（所有变量均用小写字母）

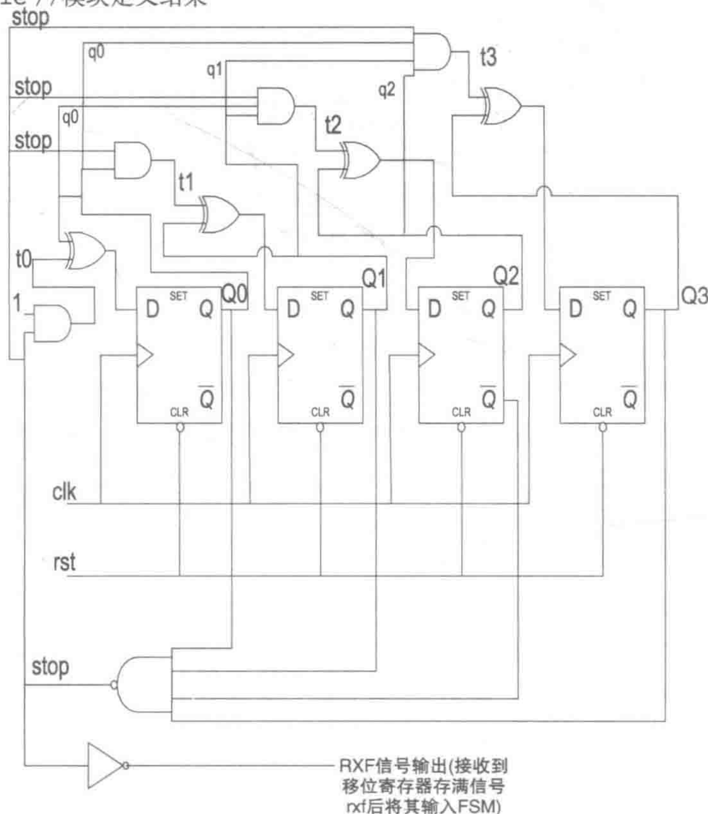
```
//定义 T 触发器
//用于构建 4 位异步计数器
module T_ FF (q, t, clk, rst);
output q;
input t, clk, rst;
reg q;
always @  (posedge clk or negedge rst)
    if (rst  = =  0)
      q < =1'b0;
    else
      q < =t^q;
endmodule

//然后是计数器模块
module divideby11 (Q0, Q1, Q2, Q3, clk, rst, RXF);
  input clk, rst; //clk 和 rst 是输入信号
  output RXF, Q0, Q1, Q2, Q3; //所有的输出信号
```

```
  wire t0, t1, t2, t3, stop; //所有 t 输入是内部线网型

//一开始先定义每一个 T 触发器
T_ FF ff0 (Q0, t0, clk, rst);
T_ FF ff1 (Q1, t1clk, rst);
T_ FF ff2 (Q2, t2, clk, rst);
T_ FF ff3 (Q3, t3, clk, rst);

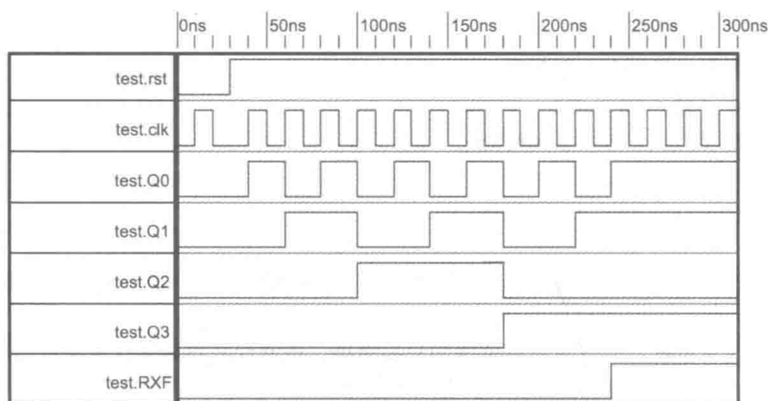
//现在用连续赋值语句 (assign) 来定义每一个输入信号 t 和其他变量的关系
assign
t0 =1'b1 & stop, //这里的赋值语句只要遵循二进制计数器的设计方法就可以了
t1 =Q0 & stop, //其实就是用与门来构建
t2 =Q0 & Q1 & stop,
t3 =Q0 & Q1 & Q2 & stop,
stop = ~ (Q0 & Q1 &~Q2 & Q3), //检测第 11 个时钟脉冲是否到达
RXF = ~stop;
endmodule //模块定义结束
```



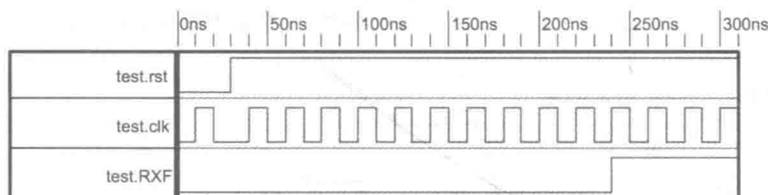
带有自动停止计数功能的4位计数器

a)

图 B. 13　a) 带有停止控制信号的 4 位计数器电路图



显示4位计数器每一位的输出



只显示向异步接收机输入信号的
4位计数器端口仿真结果
b)

图 B. 13　b) 4 位计数器仿真波形图（续）

代码 B. 4　4 位计数器 Verilog 语言描述

电路中的与非门会让仿真在第 11 个时钟脉冲到来时停止。然后通过非门将 RXF 信号拉高。RXF（寄存器存满指示）信号的作用，是通知 FSM 移位寄存器已经存满数据。此信号在数据被传输到本地数据锁存器之后会被 FSM 清零。

图 B. 13b 是仿真的波形图。

B. 9. 3　第 4 章异步接收模块的系统仿真

4. 7 节所介绍的异步数据接收系统，在这里可以进行系统仿真了。

图 B. 14 是数据接收系统的仿真结果。波形图中只含有完整模块所对应的信号，同时也将二次状态变量包含在其中，让大家对 FSM 的运行有一个清晰的概念。系统由信号 en 触发（拉高），随后 FSM（图中没有显现）会控制移位寄存器，4 位计数器和数据锁存器的操作过程。

用户可以在信号 DRY 被拉高后，读取系统接收到的数据，并通过拉高信号 ack 给予系统一个反馈。随后 FSM 会把信号 DRY（和 PD）清零，用户也可以将 ack 信号清零标志着数据传输的结束。在数据被加载到锁存器里之前，其内容是未知的（或者只能看到最后接收的那一位）。

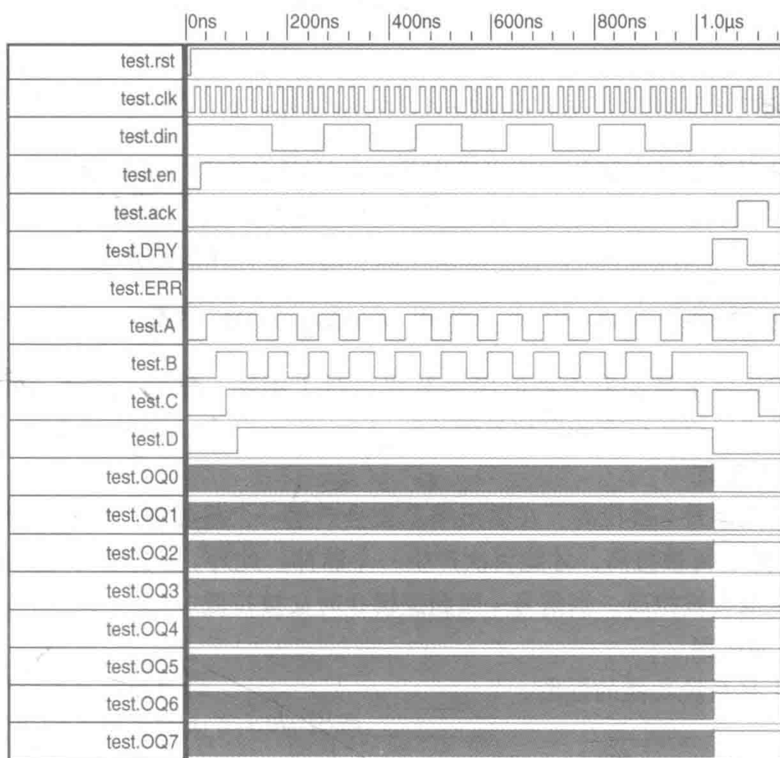


图 B.14　异步接收系统仿真结果

B.10　小结

本附录主要介绍了在 PLD 或者 FPGA 器件里，设计带有并行加载输入的同步递增和递减计数器的一些简单方法，同时还介绍了如何设计并运用带有并行加载输入的移位寄存器。

这里介绍的模块和设计方法，可以在本书中的其他例子里得到有效的运用。其中计数器和移位寄存器按位推导的方程式，可以直接转化为 Verilog HDL，使得电路设计变得十分方便。

最终，通过第 4 章介绍的异步串行接收系统来验证这里介绍的方法，并给出完整的模块设计。

附录 C 使用 Verilog HDL 仿真 FSM

C.1 概述

本附录主要是向大家介绍如何使用 SynaptiCAD 公司的 VeriLogger Extreme 仿真器，对 FSM 进行仿真，这里的 FSM 都是用 Verilog 代码进行建模。

关于 Verilog HDL 本身，在本书第 6 章 ~ 第 8 章有详细的介绍。

C.2 单脉冲同步 FSM 设计：使用 Verilog HDL 仿真

本节首先将单脉冲发生器的设计过程进行了阐述，然后编写 Verilog HDL 代码。代码风格保持尽可能的简洁易懂。

C.2.1 系统概述

只要输入信号 s 被拉高，在输出端 P 就会产生一个单脉冲。信号 s 必须首先回到逻辑 0，然后再被拉高，才能再次产生一个脉冲。此外，一个指示输出信号 L 在脉冲产生时也会被拉高，当信号 s 回到逻辑 0 时又被拉低。

C.2.2 模块框图

图 C.1 是系统功能的框图。

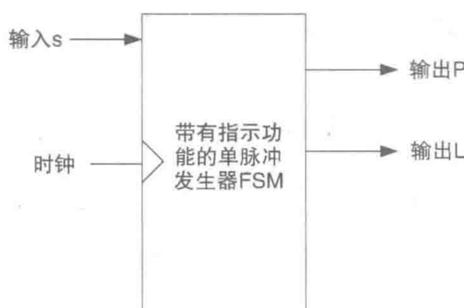


图 C.1 系统功能框图

C.2.3 状态图

FSM 框图在图 C.2 里体现。

C.2.4 状态图对应的公式

系统使用的 D 触发器公式可以从图 C.2 里直接推导出来。

$$
\begin{aligned}
A \cdot d &= s0 \cdot s + s1 \\
&= /A \cdot /B \cdot s + A \cdot /B \\
&= /B \cdot s + A \cdot /B
\end{aligned}
$$

$$
\begin{aligned}
B \cdot d &= s1 + s2 + s3 \cdot s \\
&= A \cdot /B + A \cdot B + /A \cdot B \cdot s \\
&= A + B \cdot s
\end{aligned}
$$

输出公式为：

$$P = s1 = A/B,\ L = s2 + s3 = B$$

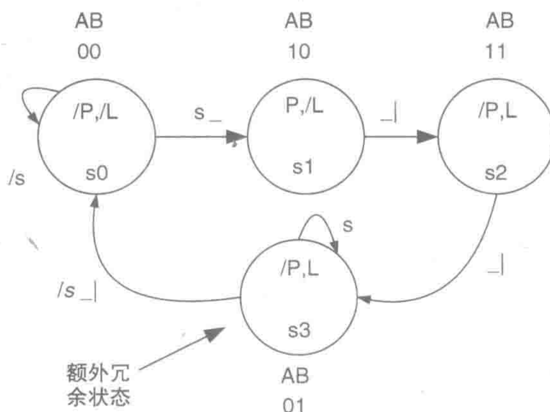


图 C. 2　系统状态框图

C. 2. 5　Verilog 描述代码

公式可以直接转换为 Verilog 代码：

```
ad = ~B & s | A & ~B,
bd = A | B & s,
P = A & ~B,
L = B;
```

这里，“与”操作符（·）被（&）所替换，“或”操作符（+）被（|）替换，“非”操作符（/）被（~）替换。每一行语句都用逗号（,）结尾，除了最后一个公式用了分号（;）。这样写的目的是为了将这些公式融合到连续赋值语句里：

```
assign
ad = ~B & s | A & ~B,
bd = A | B & s,
P = A & ~B,
L = B;
```

现在需要将 Verilog 代码补充完整。由于是用公式推导的方式进行模块设计，因此 Verilog HDL 文件用数据流模式来产生，即用事先预设的逻辑公式形成文件。

另外的方法，比如用逻辑门关系，或者用行为结构语句来表述等，在第 6 章 ~ 第 8 章有详细介绍。

当使用 Verilog 时，完整的设计可以由多个模块组成。一个模块可以含有一个或多个输入，以及一个或多个输出，这些端口可以用线网进行连接。

本例中，Verilog 文件由 3 个模块组成：

- 第一个模块是描述构成系统的 D 触发器的行为模式；
- 第二个模块描述 FSM；
- 第三个模块描述系统的测试步骤（一般被命名为“测试固件”“测试平台”或者“测试模块”）。

现在先来看第一个描述 D 触发器的模块。不管电路的行为模式有多少种，作为 D 触发器而言，其特性是标准的。将模块命名为 D_ FF，代码如 C. 1 所示：

```
1 module D_ FF (output q, input d, clk, rst);
2 reg q;
3 always @  (posedge clk or negedge rst)
4 if (rst  = =  0)
5      q< =1'b0;
6 else
7      q< =d;
8 endmodule
```

代码 C. 1　定义 D 触发器行为的模块

D 触发器的行为描述具体对应的是它输出端的行为模式。关键词 module 和 endmodule 定义了整个模块的开头和结尾。D_ FF 是模块名，括号内是 D 触发器的端口信号。这里的 q 是输出信号，d、clk 和 rst 是输入信号。关键词 output 和 input 是用来定义信号的类型的。每行开头的数字代表代码的行数，仅用作参考，它们并不跟随代码一起生成 Verilog 文件。

触发器的输出必须能够保持上一次的赋值（或者当前的值），因此输出被定义为寄存器型（reg）。需要注意的是代码的第 1、2、5、7 行每行以分号结束。

在代码第 3 行，关键词 always 后面的条件影响了代码第 4 行 ~ 第 7 行的执行。Verilog 是定义硬件电路，每一部分硬件描述必须能够并行执行。因此 always 后面 @ 符号跟随的是决定代码第 5 行 ~ 第 7 行的执行条件。根据模块的设计要求，时钟信号 clk 上面来一个上升沿（从逻辑 0 变为逻辑 1，即 posedge），或者复位信号 rst 上面出现逻辑 1 到逻辑 0 的变化（下降沿，即 negedge）时，触发语句的执行。

在代码第 4 行，出现一个 if... else 结构，它决定了代码第 5 行和第 7 行在什么情况下执行。如果复位信号 rst 是逻辑 0，那么就执行第 5 行命令，即 q < = 1'b0；否则，执行代码第 7 行，即 q < = d；这里的赋值使用了 < = 而不是 = 。对于时序语句来说非阻塞语句更为常用（具体请读者参考第 6 章的内容）。

第 5 行的赋值语句，即 q < = 1'b0；，将逻辑 0 输出到端口 q。1'b0 是 Verilog 定义单比特二进制数 0 的方法，逻辑 1 则表示为 1'b1。因此，这里的语法结构是 <比特数>'b <二进制值 0 或 1>。

第7行的赋值语句，即 q < = d，所做的事情就是将输入信号 d 赋予输出信号 q，这也是 D 触发器的行为。

代码在第8行指示模块结束。

综上所述，模块 D_ FF 定义了 D 触发器的行为。如果信号 rst（复位）是逻辑 0（触发下降沿），然后条件 if（rst = = 0）将会成立，代码第5行的赋值语句 q < = 1′b0 将执行；将触发器复位。随后，如果 rst 被拉高（复位信号被释放），每次 clk 信号出现上升沿（从0到1变化），输入端 d 的逻辑值将被赋给输出端 q。

定义逻辑电路和系统的行为的方法，在第6章和第7章都已经有详细阐述。

第二个模块（代码 C.2）是 FSM 模块，其中包含两个 D_ FF 模块，一个命名为 FFA，另一个命名为 FFB（代码第3、4两行）。这两个触发器和 FSM 相连，关联方式在定义两个触发器的括号内体现。

```
1 module fsm (input S, clk, output P, L, A, B);

2 wire ad, bd;
3 D_ FF FFA (A, ad, clk, rst);
4 D_ FF FFB (B, bd, clk, rst);

5 assign
6 ad = ~B & s | A &~B | A &~s,
7 bd=A | B & s,
8 P=A &~B,
9 L=B;
10 endmodule
```

代码 C.2　状态机模块

FSM 端口的定义体现在代码第1行的括号里，同时也标明了它们是输入还是输出。而且两个触发器的输出 A 和 B 也在其中。每个 D 触发器必须和外部门电路进行连接，因此输出信号的声明必须在这里体现。

在第2行，信号 ad 和 bd（每一个 D 触发器的 d 输入）被定义为线网型，对于 FSM 来说，它们是内部信号。代码第3行和第4行定义两个触发器，使用模块名 D_ FF，后面跟着触发器的命名（A 触发器命名为 FFA，B 触发器命名为 FFB）。

这里要注意的是括号里第一个 A 是触发器的输出，ad 是触发器的输入，clk 是时钟信号，rst 是复位信号。触发器 B 括号内的信号也是一样格式。

因此，在定义了 D 触发器 D_ FF 所需要用到的 q、d、clk 和 rst 等信号的行为描述之后，这些信号在实际应用中，需要和触发器 A 里的 A、ad、clk 和 rst 对接，对于触发器 B 就是 B、bd、clk 和 rst 等信号在 FSM 里进行布局。这些信号的前后顺序是很重要的。

代码第6行～第9行是逻辑公式，它们都是连续赋值语句，因此需要用关键词

assign 来引导，这样方便 Verilog 编译工具的识别。

连续赋值语句每一行都用逗号隔开，除了第 9 行是用分号，代表着连续赋值语句的结束。语句用“=”（阻塞语句）进行赋值，因为这里连续赋值语句的执行和顺序无关（具体原因详见第 6 章的阐述）。

第 10 行的 endmodule 标志着 FSM 的描述到此结束。

代码 C.3 是一个早期 Verilog 模块定义的版本。其中，输入和输出是在模块标题定义之后声明的，而不是标题内部。一些更早版本的 Verilog 代码习惯这样的方式，而不是代码 C.2 的方式。

```
1 module fsm (s, clk, P, L, A, B); //本行信号并没有被定义为输入或者输出

2 input s, clk; //这里定义输入
3 output P, L, A, B; //这里定义输出
4 wire ad, bd;
5 D_ FF FFA (A, ad, clk, rst);
6 D_ FF FFB (B, bd, clk, rst);

7 assign
8 ad = ~B & s | A &~B | A &~s,
9 bd =A | B &s,
10 P =A &~B,
11 L =B;
12 endmodule
```

代码 C.3 “旧版”模块代码风格

目前讨论的两个模块 DFF 和 FSM 是构成系统 FSM 所需的部分。然而，为了测试 FSM 是按照设计初衷来运行的，系统还需要另一个模块，这就是测试平台模块。

C.3 测试平台和其存在的目的

图 C.3 是关于测试平台和 FSM 之间连接的系统框图，从图中可以看出测试平台向 FSM 提供测试信号。测试平台的输出信号（s、rst 和 clk）是用来测试 FSM 的状态转换过程，即整个系统的功能。

测试流程是根据状态图里状态的转换设计的，目的在于将系统可能面对的所有情况全部覆盖。加载到 FSM 并验证其功能的信号顺序是由测试平台来决定的。相对而言，第 6 章和第 7 章是向大家介绍如何产生更加有效的测试信号。

代码 C.4 是测试平台的代码。这是一种很直接的定义测试平台的方式，非常易于理解。第 6 章到第 8 章里有介绍其他的代码风格。

```
//测试平台
1 module test;
```

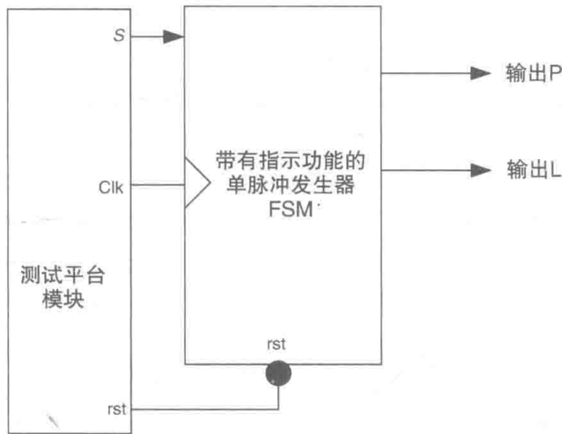


图 C.3　测试平台和 FSM 的连接

```
2 reg s, clk, rst;
3fsmsingle_ pulse (s, clk, rst, P, L, A, B);

4 initial
5 begin
6     $dumpfile ( “single_ pulse.vcd”);
7     $dumpvars ;
  //初始化输入信号
8      s =0;
9     rst =0;
10     clk =0;    //clk 时钟信号在一开始通常为低
  //FSM 停在状态 s0，因为此时复位信号仍然有效
11     #10clk = ~clk;
12     #10clk = ~clk;
  //释放复位信号，FSM 仍然在 s0
13     #10rst =1;
14     #10clk = ~clk;
15     #10clk = ~clk;
  //将信号 s 拉高，进入状态 s1
16     #10 s =1;
17     #10clk = ~clk;
18     #10clk = ~clk;
  //下一个时钟脉冲到来时进入 s2
19     #10clk = ~clk;
20     #10clk = ~clk;
```

```
  //再下一个时钟脉冲到来时进入 s3
21     #10clk = ~clk;
22     #10clk = ~clk;
  //下一个时钟脉冲到来时停在状态 s3，因为信号 s 仍然为逻辑 1
23     #10clk = ~clk;
24     #10clk = ~clk;
  //下一个时钟脉冲到来时让信号 s 变为逻辑 0，使得 FSM 回到 s0
25     s =0;
26     #10clk = ~clk;
27     #10clk = ~clk;
  //使用 reap 循环在 4 个时钟脉冲的时间段内让整个过程再走一遍
28     #10 s =1;
29     repeat (8)
30     #10clk = ~clk;
  //回到状态 s3
31     #10 s =0;
32     #10clk = ~clk;
33     #10clk = ~clk;
  //回到状态 s0
  //完成仿真
34     #10 $finish;
35     end
36 endmodule
```

代码 C.4 测试平台代码

代码从第 1 行开始，命名为 test。模块没有，也没必要有任何输入信号，除了代码第 2 行所列出的连接 FSM 的信号之外。这些信号分别是 s、clk 和 rst，它们都被定义为寄存器型，用关键词 reg 引导。定义成寄存器型的目的在于整个测试代码运行过程中，这些信号的赋值需要被一直保持。

FSM 模块的建模在代码第 3 行，命名为 single_ pulse。

后面括号里信号排列的顺序必须和定义 FSM 时所用的顺序一样。

随后的第 4 行 ~ 第 33 行代码是将输入信号加载到 FSM，测试状态的转换（以及输出）是否正确。测试顺序是根据状态图来制定的，按照一定的顺序加载输入信号并不断地变化，以达到将 FSM 完整地测试一遍的目的。注释符（以“//”开始的双斜线）后面的文字提示了测试步骤。

通过学习状态图（见图 C.2）和测试程序（见代码 C.3），读者应该试着去理解如何设计系统的测试步骤。

从代码第 4 行的initial 开始到第 35 行的end 结束，定义了一个完整的模块初始化过程。本书的第 6 章和第 7 章对其有更加详细的讨论。

整个initial模块里，模块的输出信号在第8、9和10行被初始化，一开始它们的值均为逻辑0。对于系统时钟clk，一开始也是逻辑0，因此时钟信号上面的任何变化都只能是从0变为1。

代码第11行，时钟信号被拉高，第12行又被拉低。这就是产生时钟脉冲的方法。~clk只是将时钟信号clk的逻辑进行了反转而已。

测试代码第11行和第12行在这里的作用，是确保在复位信号rst=0的情况下，FSM将保持在状态s0（参考图C.2）。

复位信号rst在第13行被拉高，但是大家要注意其代码格式为：#10rst=1。这里#10的重要性在于，信号rst被拉高释放之前将整个测试过程延迟10个时间单位。实际的延迟时间可以通过具体的时钟周期计算并量化，这里假设仿真被延迟了10ns。现在大家看到的情况是在延迟之前（0ns）的时刻，信号s、rst和clk的值都为逻辑0，10ns后，rst变为逻辑1。通过这种方法，可以将需要测试FSM功能的信号按照一定的次序不断地进行变化。那么代码第11行和第12行的时钟信号，可以看作是一个10ns宽度的时钟脉冲。

代码第14行和第15行产生了另一个时钟脉冲（clk信号从0到1再到0）。然而，此时信号s为逻辑0，所以FSM仍然维持在状态s0。

在代码第16行，信号s被拉高，FSM根据时钟的脉冲从s0进入s3（代码第17行~第24行）。

在代码第23行和第24行，FSM应该保持在s3，因为此时s信号仍然有效。

在第25行，s=0，FSM在下一个时钟脉冲到来时回到状态s0（代码第26、27行）。

在第28行，信号s再次被激活，代码第29、30行用repeat循环语句来产生4个周期的时钟脉冲，FSM也在其过程中从s0再次进入s3。这两行代码的目的主要还是产生一定数量的时钟周期。在代码第31行，信号s回到逻辑0。随后的两条语句驱动FSM回到状态s0。最终FSM的仿真在第34行用关键词 $finish结束（这里可以用 $stop代替）。

整个Verilog文件通过编译和仿真来验证设计功能的完整性。如果设计过程中有某些错误（有可能是输入代码时的语法错误，即拼写错误），必须要首先全部排除，然后再次编译和仿真。

当按下编译按钮时（见图C.4），会弹出一个文件被修改的对话框，单击"yes"之后，仿真会在错误全部被修正后重新启动。

这里FSM所用的仿真工具是SynaptiCAD公司出的VeriLogger Extreme仿真器。图C.4是仿真工具的软件界面，其中Verilog代码在右上方，仿真波形在左下方。

C.4　使用SynaptiCAD公司的VeriLogger Extreme仿真器

首先运行软件：

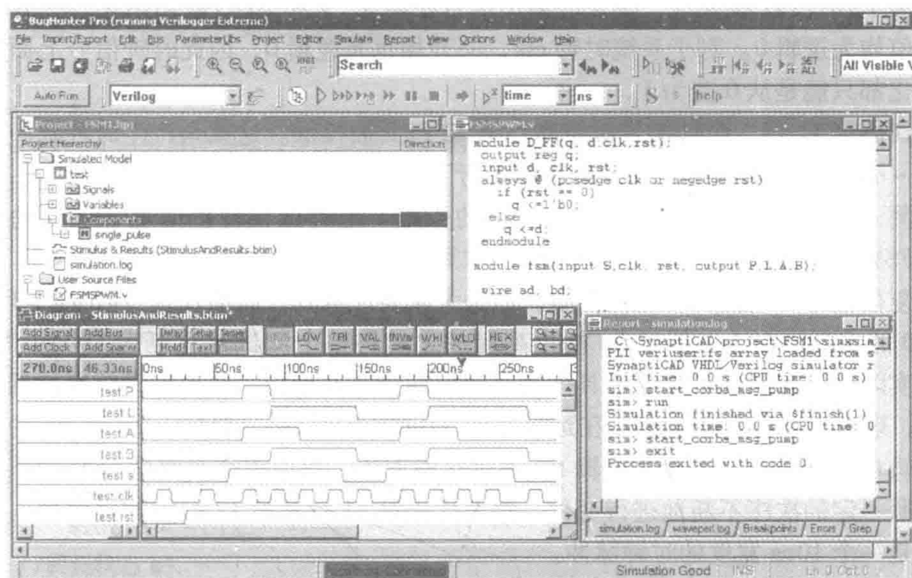


图 C.4　带有 BugHunter 图形纠错功能的 VeriLogger Extreme 仿真工具界面

• 按照“开始”→SynaptiCAD→Simulation Debug→VeriLogger Extreme + BugHunter 的顺序选择菜单并打开软件。

• 如果需要看软件说明，可以按照 Help→BugHunterVeriLogger Manual 的菜单顺序启动软件的帮助文档，获得如何操作软件的完整的用户说明。

• 目录 Help→Tutorials→Basic Verilog Simulation 可以引导大家打开一个教学文档，其中有对软件图形界面和生成测试固件等详细的功能说明。

然后生成一个工程用来保存需要仿真的文件：

• 在软件中选择 Project→New Project 菜单，并打开一个新建工程（New Project Wizard）对话框（见图 C.5）。

• 在工程名（Project Name）一栏，输入 FSM1，然后单击“Finish”按钮，便生成了一个名为 FSM1. hpj 的工程。

复制或者产生/添加源文件到工程目录：

• 如需要复制源文件，鼠标右键单击“User Source Files Folder”，从下拉菜单中选择“Copy HDL Files to Source File Folder”，随后会弹出一个对话框（见图 C.6）并用浏览功能从光盘里找到 FSMSPWM. v。

• 也可以从顶部菜单选择 Editor→New HDL File 打开一个编辑对话框，创建一个文件并命名为 FSMSPWM. v。随后将代码输入创建的文件并保存。再单击“User Source Files Folder”，选择“Add HDL Files to Source File Folder”将保存的文件添加到工程里。

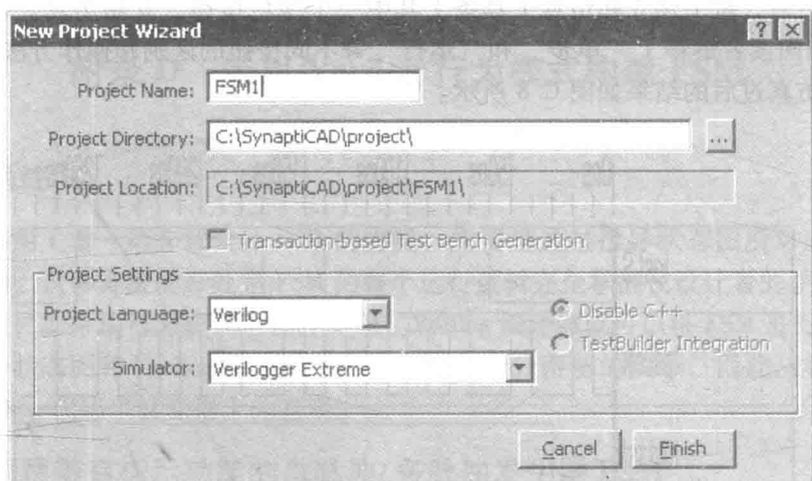


图 C.5　新建工程对话框

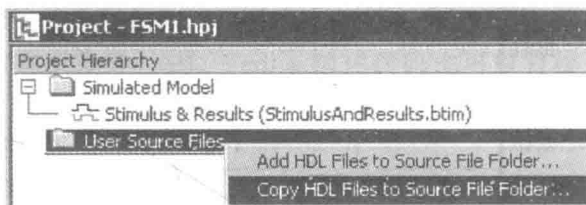


图 C.6　如何将代码 C.1 ~ C.4 复制到工程目录下

建立工程和仿真代码：

- 首先，在软件仿真菜单条上单击黄色的“建立”按钮，或者在菜单项里选择 Simulate →Build（见图 C.7）进行工程编译。



图 C.7　建立工程和仿真代码的工具条

- 建立工程包括编译源文件，在工程窗口里形成设计的构架，并对顶层的信号和变量设置监控。在每次仿真之前，系统都会自动先运行一遍建立工程的流程。但是软件会留出一个专门的建立工程按钮，这样用户不必等待整个仿真过程结束，就可以看到工程构架。工程建立完成之后，用户可以设立工程的顶层文件，也可以使用工程构架菜单，向仿真界面添加新的信号以便观察。监视信号就是那些图 C.4 里左下方“Stimulus and Results”图表里列出的信号。
- 在“Report”一栏检查工程建立过程中产生的语法错误。

• 然后，按下建立和仿真工具条上其中一个绿色按钮，开始仿真。软件在线帮助文档向读者解释了“单步”和“运行”等不同按钮的区别和操作方法。

• 仿真过后的结果如图 C.8 所示。

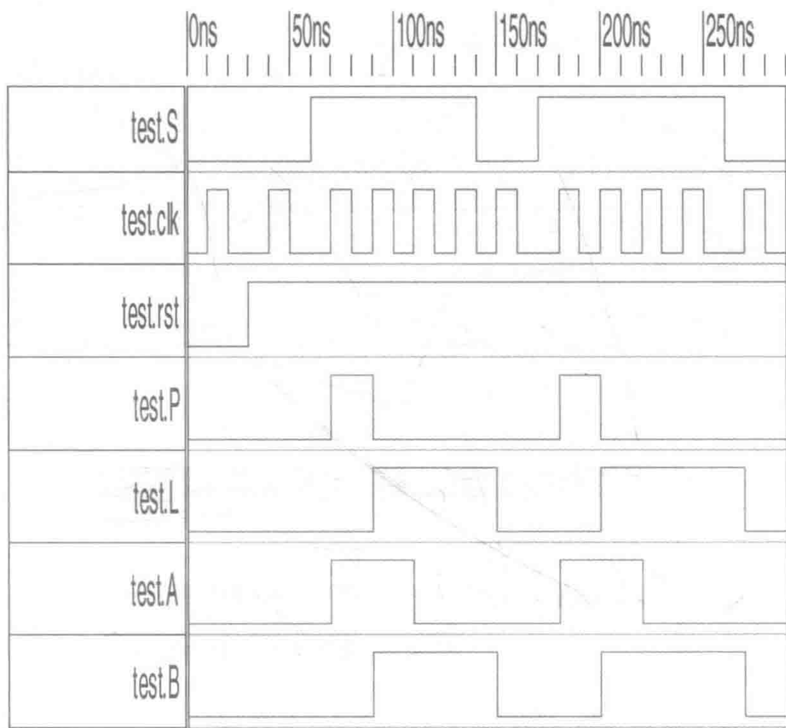


图 C.8 Verilog 仿真波形图

• 如果希望从垂直方向观察仿真结果，可以在菜单栏上选择 Options →Drawing Preferences 打开一个对话框。在“Edge Display”一栏里对“Straight”按钮打钩。

C.5 小结

本章介绍了运用 Verilog 语言设计 FSM 的众多方法中的一种。其他方法读者可以参考第 6 章 ~ 第 8 章的内容。同时里面还包含了指导大家如何运用 SynaptiCAD 公司的 VeriLogger Extreme 仿真器去验证 FSM 设计方案。

由于状态图中涵盖了设计系统所需的所有信息，因此介绍的设计方法很容易掌握。这些信息可以从公式中进行“提取”以便综合设计方案。随后的仿真目的在于验证设计的正确性。本书用到的最新版本的软件，可以从 SynaptiCAD 公司的官网 http：//www. syncad. com 下载。读者必须关注软件的实时更新，并从上述网站获得最新的版本。

附录 D　运用 Verilog 行为模式构建 FSM

D.1　概述

本书第 1 章 ~ 第 5 章设计 FSM 的方法，主要是通过推导状态图所对应的公式来完成的。这种方法的好处是 FSM 的整个运行逻辑完全掌握在设计者的手中。

不过，如果用行为模式来设计 FSM，Verilog 编译器就可以将 FSM 进行优化。

由于状态图和 Verilog 行为描述之间存在着十分密切的关系，因此从状态图转化为 Verilog 代码也就变得十分直观。

D.2　回顾带有指示功能的单脉冲/多脉冲发生器 FSM

系统中带有两个输入：信号 s 用于启动系统工作；信号 x 用来选择系统输出单脉冲，还是多脉冲。在单脉冲模式下，输出信号 L 用来指示单脉冲的产生。在多脉冲模式下，信号 L 不起任何作用。图 D.1 是系统的状态图。

除了直接从状态图推导公式，Verilog 描述代码也可以直接从状态图中得出，如代码 D.1 所示。

```
//状态机行为描述
1 module pulsar (s, clk, rst, P, L, ab);
2    input s, clk, rst;
2    output [1: 0] ab, P, L;
3    reg [1: 0] state, P, L;

4    parameter s0 =2'b00, s1 =2'b01, s2 =2'b11, s3 =2'b10;

//现在（从状态图中）定义 FSM 的状态序列
5 always@  (posedge clk or negedge rst)
6 if ( ~ rst)
7   state  < = s0;
8 else
9 case (state)
10    s0: if (s) state  < = s1; else state  < = s0;
11    s1: state  < = s2;
12    s2: if ( ~ x) state  < = s3; else state  < = s1;
13    s3: if ( ~ s) state  < = s0; else state  < = s3;
14 endcase

//现在定义各个状态的输出
```

```
15 always @  (state)
16 case (state)
17   s0: begin P =1'b0; L =1'b0; end
18   s1: begin P =1'b1;
19          L= (state == s1) & ~x; //米利输出
20   end
21   s2: begin L= (state == s2) & ~x; P =1'b0; end
22   s3: begin P =1'b0; L= (state == s3) & ~x; end
23 endcase
24 assign ab = state;
25 endmodule
```

代码 D.1　状态图的 Verilog 描述

现在可以来做一个状态图 D.1 和状态机行为的比较：

代码 D.1 中，第 5 行 ~ 第 14 行基于状态和输入信号的不同值，来定义了系统状态切换的顺序。代码第 15 行 ~ 第 24 行将系统的输出信号，在每一个状态所对应的值进行了定义。这两组时序语句是同时发生的（即并行执行）。

代码第 1 行和第 2 行定义了模块的名称和信号名，随后定义了输入、输出以及信号的类型。

输出是在第 2 行声明的。2 比特位宽的数组信号［1:0］ab 是用来显示 FSM 的状态（见图 D.2）。第 3 行定义了状态数组［1:0］state，目的是用来监视 FSM 的当前运行情况。同时输出信号 P 和 L 也在这一行被定义为寄存器（reg）型变量。第 4 行定义了每一个状态的二次状态变量。注意这里定义的二次状态变量的值和状态图 D.1 里的值是统一的。

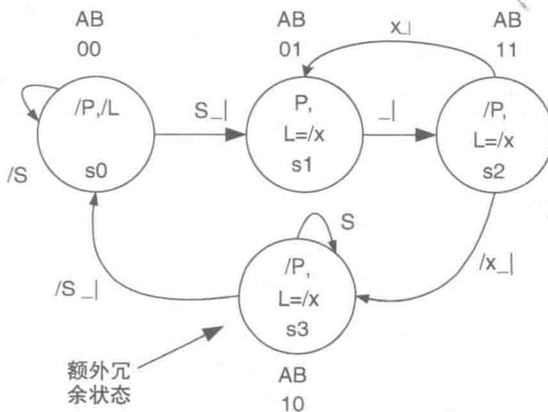


图 D.1　带有指示功能的单脉冲 FSM

第5行的always语句定义了后面case语句（第9~14行）被调用的条件。第6行是判断rst信号的变化是否为从1变为0；否则else后面的case语句将立刻触发（第10~14行）。

这里的case语句定义了FSM可能存在的4个状态，以及各个状态之间的切换条件。case语句构架在这里的运行机制是选择一个当前态，然后根据输入条件来判断当前态的下一个状态。起初的复位完成之后，FSM将会处于s0（2′b00），此时if语句开始判断输入信号s的值。

如果输入信号s为逻辑1，那么下一个状态则为s1，否则仍为s0。

在下一个时钟的上升沿（posedge）到来时，case语句会被再次激活。如果此时系统在状态s1，代码第11行将会把系统送往s2。

随后的一个时钟上升沿，代码第12行将为系统在s3和s1之间做出选择，如果x=0，则系统进入s3；如果x为逻辑1，则回到s1。在代码第13行，程序会测试输入信号s，如果s=0，FSM会回到s0，否则会停留在s3。

整个过程就是一个用case语句来决定状态切换的顺序，并用if语句判断输入条件，然后决定下一状态的思路。当没有输入条件提供判断时，程序会直接让系统进入下一状态。

输出信号是用另一个case语句模块，根据系统所处不同的状态来定义的。注意在第19行，米利型输出L在状态s1时是采用如下方式定义的：L=（state==s1）&~x，这意味着L的结果受到输入信号x（x=0）和系统在状态s1的双重影响，即L=s1·/x。这和状态图中标记的结果是一致的。

因此，只有当x=0且系统在状态s1的情况下，L才为逻辑1，否则它将被忽略。而输出P在状态s1被赋值为逻辑1。

对于其他状态，case语句针对状态图D.1判断输出信号赋值的方式是一样的。

通过比较两个always行为模块和状态图D.1，大家可以看出Verilog描述和状态图之间存在着很强的关联性。

这样，在设计FSM时，就不需要事先推导公式并规划硬件电路了。对于FSM来说，这其实就是一种“终端的行为和序列”所形成的效应。

代码D.2是系统的测试程序，产生方法和本书其他章节介绍的测试平台一样。

```
//定义测试平台固件
module test'
reg s, clk, rst;
wire [1: 0] ab;
wire p, l;
pulsaruut (s, clk, rst, p, l, ab);
initial
  begin
    rst =0;
```

```
clk =0;
s =0;
x =0;
#10 clk = ~clk;
#10 clk = ~clk;
//处于状态 s0。
#10 rst =1;
#10 s =1;
#10 clk = ~clk;
#10 clk = ~clk;
//进入状态 s1。
#10 clk = ~clk;
#10 clk = ~clk;
//进入状态 s2
#10 clk = ~clk;
#10 clk = ~clk;
//停留在状态 s2
#10 s =0;
#10 clk = ~clk;
#10 clk = ~clk;
//回到状态 s0
//让 x =1，产生多个脉冲
#10 x =1;
#10 s =1;
#10 clk = ~clk;
#10 clk = ~clk;
//进入状态 s1
#10 clk = ~clk;
#10 clk = ~clk;
//进入状态 s2
#10 clk = ~clk;
#10 clk = ~clk;
//再次进入状态 s1
#10 clk = ~clk;
#10 clk = ~clk;
//仍然在状态 s1
#10 clk = ~clk;
#10 clk = ~clk;
//再次到 s2
```

```
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10  x =0; //准备停止多脉冲模式
    #10 clk = ~clk;
    #10 clk = ~clk;
    //到达状态 s3
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10  s =0;
    #10 clk = ~clk;
    #10 clk = ~clk;
    //回到状态 s0
    #10  $stop ;
    end
endmodule
```

代码 D. 2 测试平台程序

图 D. 2 是系统的测试波形图。数组 ab 代表每个时钟上升沿到来时系统所处的状态，大家可以将其和图 D. 1 进行比对。

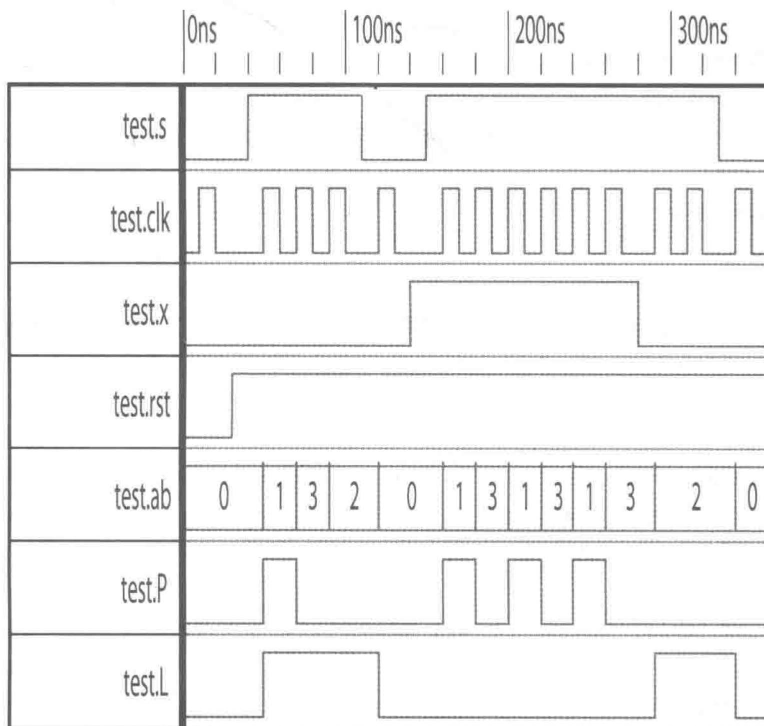


图 D. 2　FSM 仿真波形图

根据波形图产生的系统运行顺序，对照状态图验证系统每一步状态切换是否正确。可以看到单脉冲和多脉冲两个模式都能够正常工作：

当 x=0，FSM 产生单脉冲；

当 x=1，FSM 在 s1 和 s2 之间反复切换时产生多个脉冲，最终在 x=0 时回到状态 s0。

D.3　5.6 节中存储芯片测试系统

同样，系统可以对照图 5.15 直接编写 Verilog 描述代码，结果如代码 D.3 所示

```
module
memory_ tester_ state_ machine (clk, rst, st, fab, full, RC, P, CS,
RD, WR, OK, ERROR, abc);
    input clk, rst, st, fab, full;
    output [3: 0] abc, RC, P, CS, RD, WR, OK, ERROR;

    reg [3: 0] state, RC, P, CS, RD, WR, OK, ERROR;
//二次状态变量赋值
    parameter s0 =4'b0000, s1 =4'b1000, s2 =4'b1010, s3 =4'b0010,
            s4 =4'b0110, s5 =4'b1110, s6 =4'b1100, s7 =4'b1101,
            s8 =4'b1001, s9 =4'b1011, s10 =4'b0100;
//状态机序列

always @  (posedge clk or negedge rst)
  if (~rst)
    state=s0;
  else
  case (state)
    s0: if (st) state  < = s1; else state  < = s0;
    s1: state  < = s2;
    s2: state  < = s3;
    s3: state  < = s4;
    s4: state  < = s5;
    s5: state  < = s6;
    s6: if (fab) state  < = s7; else state  < = s10;
    s7: state  < = s8;
    s8: if (full) state  < = s9; else state  < = s1;
    s9: state  < = s9;
    s10: state  < = s10;
  endcase
//每个状态的输出
```

```
always @  (state)
  begin
    {RC, P, CS, RD, WR, OK, ERROR} =7'b0011100;
   case (state)
   s0: begin assign RC =1'b0;
           P =0;
           CS =1;
           RD =1;
           WR =1;
           OK =0;
           ERROR =0;
     end
  s1: begin assign RC =1'b1;
           CS =1'b0;
           P =0
     end
  s2: begin WR =1'b0; end
  s3: begin CS =1'b0;
         WR =1'b1;
     end
  s4: begin end
  s5: begin RD =1'b0; end
  s6: begin end
  s7: begin RD =1'b1; end
  s8: begin CS =1'b1;
         P =1'b1;
     end
  s9: begin OK =1'b1;
         P =0
     end
  s10: begin ERROR =1'b1; end
 endcase
end

assign abc =state;
endmodule

module test;
```

```
   reg st, clk, rst, fab, full;
   wire [3: 0] abc;
   memory_ tester_ state_ machineuut (clk, rst, st, fab, full, RC, P, CS,
RD, WR, OK, ERROR, abc);

   initial
     begin
       rst =0;
       clk =0;
       st =0;
       fab =0;
       full =0;
       #10 rst =1;
       #10 st =1;
       #10 repeat (14)
         #10 clk = ~clk;
       #10 rst =0;
       #10 rst =1;
       #10 repeat (10)
       #10 clk = ~clk;
       #10 fab =1;
       #10 repeat (16)
       #10 clk = ~clk;
       #10 clk = ~clk;
       //#10 rst =0;
       //#10 rst =1;
       #10 full =1;
       #10 repeat (20)
       #10 clk = ~clk;
       #10 $finish
     end
   endmodule
```

代码 D. 3　存储芯片行为描述代码

根据上述代码得出的测试结果波形图如图 D. 3 所示。

数组 abc 的值和 FSM 的二次状态变量是一一对应的。它们的赋值方式符合单位距离编码，在图 D. 3 中显示的格式为十六进制。

FSM 在仿真波形图中移动的顺序是 s0 （0）、s1 （8）、s2 （A）、s3 （2）、s4 （6）、s5 （E）、s6 （C），然后 s10 （4） 报错。随后进行复位回到 s0，并在到达 s8 （9） 之后再次循环进入 s9 （B）。

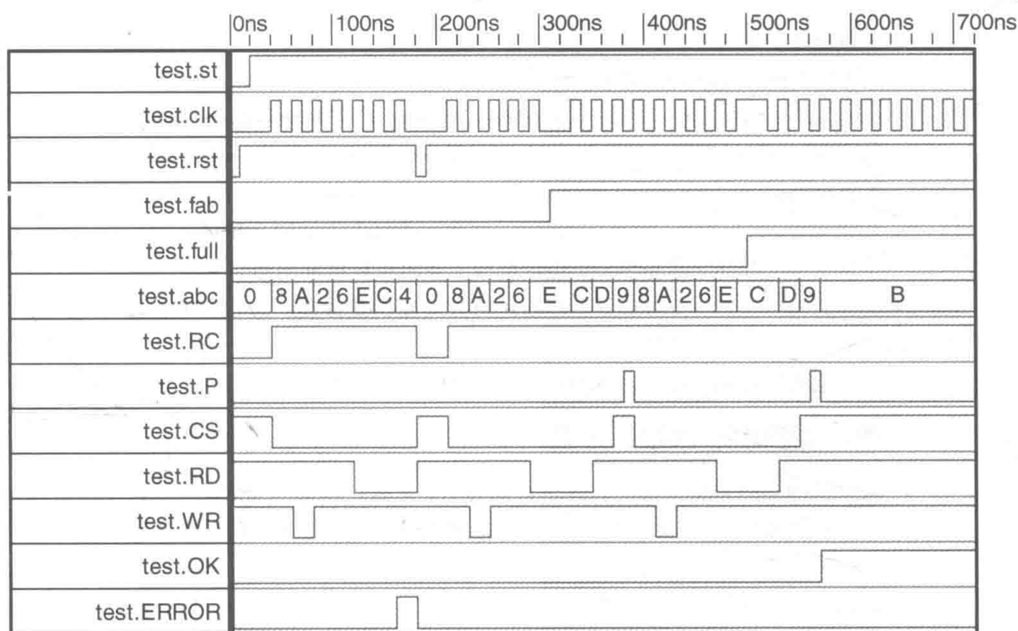


图 D.3 系统仿真波形图

存储空间的写循环（8，A，2）后面是读循环（E，C），完整的一次就是（8，A，2，6，E，C，D）。

D.4 小结

应用行为描述的一大好处就是直接从状态图中提取信息进行编程。可以发现状态图的结构和 Verilog 代码构架之间存在一对一的对应关系，这也从侧面反映了状态图提取的信息既可以用来推导公式，也可以直接完成系统设计（行为描述）。

两种描述方法都可以在计算机上执行并验证，这些已经在教学领域得到广泛的运用。

……国际视野　科技前沿……

国际信息工程先进技术译丛

《基于FSM和Verilog HDL的数字电路设
《移动协议与切换优化：设计、评估与应用
《全IP网络融合》
《不确定性理论与多传感器数据融合》
《无线通信系统中的定位技术与应用》
《基于大数据的商务智能分析》
《3GPP网络中的IPv6部署：从2G向LTE及未来移动宽带的演进》
《MIMO无线网络手册》
《可重构无线电系统的网络架构和标准》
《声学显微镜与超分辨率成像理论及应用》
《构建基于IPv6和移动IPv6的物联网：向M2M通信的演进》
《虚拟网络——下一代互联网的多元化方法》
《下一代融合网络理论与实践》
《认知视角下的无线传感器网络》
《移动云计算：无线、移动及社交网络中分布式资源的开发利用》
《Android系统安全与攻防》
《内容分发网络》
《计算机网络仿真OPNET实用指南》
《移动无线信道》（原书第2版）
《LTE-Advanced：面向IMT-Advanced的3GPP解决方案》
《声学成像技术及工程应用》
《LTE/SAE网络部署实用指南》
《认知无线电通信与组网：原理与应用》
《网络性能分析原理与应用》
《云连接与嵌入式传感系统》
《IP地址管理原理与实践》
《自组织网络：GSM,UMTS和LTE的自规划、自优化和自愈合》
《实现吉比特传输的60GHz无线通信技术》
《LTE自组织网络（SON）：高效的网络管理自动化》
《UMTS中的LTE：向LTE-Advanced演进》（原书第2版）
《UMTS中的WCDMA－HSPA演进及LTE（原书第5版）
《无线传感器及执行器网络》
《认知无线电网络》
《网络融合——服务、应用、传输和运营支撑》
《UMTS中的LTE:基于OFDMA和SC-FDMA的无线接入》
《吉规模集成电路互连工艺及设计》
《高性能微处理器电路设计》

WILEY





机械工业出版社微信公众号



机械工业出版社E视界



上架指导　工业技术／数字电路设计

ISBN 978-7-111-53292-7

定价：120.00元

[General Information]
书名=基于FSM和Verilog HDL的数字电路设计=FSM based digital design using verilog HDL
作者=（英）彼德?明斯，伊恩?艾利奥特著
页数=361
SS号=14051791
DX号=
出版日期=2016.05
出版社=机械工业出版社